INTERFACING A REASONER WITH HETEROGENEOUS SELF-CONTROLLING SOFTWARE

A Dissertation Presented

by

Jakub Jerzy Moskal

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in the field of

Computer Engineering

Northeastern University
Boston, Massachusetts

April, 2011
© Copyright 2011 by Jakub Jerzy Moskal
All Rights Reserved
Abstract

Software Defined Radio (SDR) is a radio in which some or all of the physical layer functions are software defined, i.e. radio’s operating functions, such as modulation scheme or frequency, are implemented with the use of software processing, as opposed to being implemented in hardware. Existing SDR architectures differ in the scope and the partition of the offered functionality between the reconfigurable software and the hardware components. Thus, each SDR offers a different set of adjustable and observable parameters (also called *knobs* and *meters*).

Cognitive Radio (CR) is a term to describe a radio communication paradigm which takes advantage of the SDR architecture and allows for dynamic changes of radio’s operational behavior in order to achieve a variety of goals, including interoperability, performance optimization, opportunistic use of resources and others. Although CR could be implemented in many ways, employing different artificial intelligence and optimization algorithms, in this work we focus solely on the knowledge-based CR architecture which utilizes an inference engine to provide the cognitive capability. This approach separates the knowledge about the radio domain from the cognitive algorithms, allowing domain experts to express their knowledge in an implementation-independent fashion by specifying ontologies and rules.

Existing CR engines interface SDR platforms relying on APIs that are radio-specific and sometimes platform-dependent. So far a standard SDR API has not been established. As a consequence, the CRs are bound to a specific type of SDR, they are not reusable and are unable to adapt to changes in the SDR functionality without recoding. The lack of a standard CR architecture and a standard API for interfacing reasoners with the SDR’s parameters (*knobs* and *meters*) is the main problem of implementing CR according to the knowledge-based approach.

We propose a Cognitive Radio Framework (CRF) architecture, which aims to meet the following requirements: 1) support reusability of expert knowledge, 2) be SDR-independent, 3) be platform-independent, and 4) be able to adapt to changes in the
domain without recoding. Instead of using a domain-specific API to interface SDRs, the CRF architecture allows the reasoner to interface an SDR using a thin and generic API. This architecture is capable of maintaining the interface with heterogeneous SDRs that change their APIs without the need of recoding the reasoner API due to the fact that the interface information is provided dynamically. Moreover, since the CR can be viewed as a subclass of self-controlling software, the domain-adaptable CRF architecture can be utilized in other domains where the self-controlling software paradigm could apply, for instance, in radar tracking.
Contents

Abstract ii

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Software Defined Radio .......................... 1

1.2 Cognitive Radio .................................. 3

1.2.1 Cognition Cycle ................................ 3

1.2.2 Definition ....................................... 6

1.2.3 Applications of CR .............................. 7

1.3 Implementing CR ................................... 7

1.3.1 Knowledge Based Implementation of CS ....... 9

1.4 CR and self-controlling software .................. 11

2 Problem Formulation 13

2.1 Introduction ........................................ 13

2.2 Requirements ....................................... 14

2.3 Literature Review .................................. 15

2.3.1 SDR API ........................................ 16
3 Background Information

3.1 Knowledge Representation ........................................... 48
3.2 Ontology matching ..................................................... 50
  3.2.1 Representing and processing alignments ......................... 53
3.3 Knowledge exchange .................................................... 56
  3.3.1 Agent Communication Languages and CR networks ............... 59
3.4 CORBA Middleware ..................................................... 60

4 Cognitive Radio Framework (CRF)

4.1 Architecture ......................................................... 62
  4.1.1 Message Layer .................................................... 64
  4.1.2 Cognitive Layer .................................................. 66
  4.1.3 LiveKB .......................................................... 67
4.2 LiveKB Design ......................................................... 68
  4.2.1 Motivation ........................................................ 68
  4.2.2 Enabling generic API ............................................ 69
  4.2.3 Ideal design of LiveKB ......................................... 71
  4.2.4 Feasible design of LiveKB ..................................... 72
  4.2.5 Generating an IDL Ontology ................................... 75
  4.2.6 Generating bridge axioms ..................................... 87
4.2.7 Generating Invoker ................................................. 99
4.3 CRF Interfaces ......................................................... 106
   4.3.1 LiveKB API ....................................................... 107
   4.3.2 Message Layer APIs ........................................... 109
   4.3.3 Reasoner API ................................................... 111
4.4 Choice of middleware ............................................... 113
   4.4.1 Web Services .................................................... 114
   4.4.2 Web Services and the CRF .................................... 115
4.5 Dynamic change of the knowledge base ............................. 116
   4.5.1 Rules and events ............................................... 117
4.6 Behavioral description ............................................... 118
   4.6.1 Initialization .................................................. 119
   4.6.2 Runtime ....................................................... 119
4.7 Roles and artifacts .................................................. 121
4.8 Implementation ..................................................... 123
   4.8.1 Implementation language .................................... 123
   4.8.2 Reasoning component ........................................ 123
   4.8.3 Reasoner API ................................................. 124
   4.8.4 LiveKB component .......................................... 125
   4.8.5 Bridge axioms ............................................... 125
   4.8.6 Reliable Data Transfer (RDT) ............................... 125
5 Evaluation ............................................................. 128
   5.1 CRF demonstration ................................................ 128
      5.1.1 SDR platform ................................................. 129
      5.1.2 Implemented Scenario ..................................... 130
## List of Tables

2.1 SDR API approach and its support for the desired features of the CR architecture ........................................... 18
2.2 The Standard API approach and its support for the desired features of the CR architecture ........................... 29
2.3 Features of the SCA-based design and the requirements for the CR architecture ............................. 35
2.4 OBR architecture and its support for the desired features of the CR architecture ............................... 39
2.5 XG architecture and its support for the desired features of the CR architecture ................................. 43
2.6 Related work and the features support ............................................................................................................ 46

4.1 Correspondence between IDL primitive types and XML Schema types ............................................. 76
4.2 IDL constructs supported by automatic translation to OWL ............................................................ 81

5.1 Features of CRF and the requirements for the CR architecture .............................................................. 141
5.2 Comparison of developer’s effort required to adjust to changes in different scenarios .............................. 144
5.3 Comparison of execution steps ....................................................................................................................... 145
5.4 Comparison of run-time complexity ............................................................................................................ 146
5.5 Comparison of requirements ..................................................................................................................... 146
5.6 IDL TBox triples produced by LiveKB that are not part of the domain ABox ............................................ 147
6.1 Implementation effort expressed in lines of code ......................................................................................... 152
# List of Figures

1.1 Software Defined Radio (SDR) architecture ...................................... 2
1.2 Observe-Orient-Decide-Act (OODA) feedback loop ................................ 4
1.3 iCR – ideal cognition cycle designed by Mitola ................................. 5
1.4 Abstract view of a Cognitive Radio architecture .............................. 8
1.5 Architecture of a knowledge-based Cognitive Radio .......................... 10
1.6 Cognitive Radio viewed as a special case of a control theory-based software model 12

2.1 Visual representation of the problem formulation ............................... 14
2.2 The SDR API approach to interfacing various radio platforms ............ 16
2.3 The Standard API approach to interfacing various radio platforms ....... 19
2.4 Using XML-API approach to interface cognitive engine and various radio platforms 21
2.5 The Software Communications Architecture (SCA) ............................ 31
2.6 The Open Source Cognitive Radio (OSCR) architecture ..................... 34
2.7 The Ontology Based Radio (OBR) architecture ............................... 38
2.8 Abstract view of an XG-enabled device ......................................... 41
2.9 Unified Link Layer API architecture ............................................. 44

3.1 Architecture of a knowledge representation system based on Description Logics 49
3.2 Example of using ontology matching to generate a data instance translator 51
3.3 An example of using ontology matching to generate a mediator ........... 51
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>FIPA Query Interaction Protocol</td>
<td>58</td>
</tr>
<tr>
<td>3.5</td>
<td>Client-Server communication using CORBA middleware</td>
<td>60</td>
</tr>
<tr>
<td>4.1</td>
<td>The Cognitive Radio Framework (CRF) architecture</td>
<td>63</td>
</tr>
<tr>
<td>4.2</td>
<td>Types of messages distinguished in the OBR architecture</td>
<td>64</td>
</tr>
<tr>
<td>4.3</td>
<td>Types of messages distinguished in the CRF architecture</td>
<td>65</td>
</tr>
<tr>
<td>4.4</td>
<td>Accessing SDR using the LiveKB component</td>
<td>70</td>
</tr>
<tr>
<td>4.5</td>
<td>Ideal design of LiveKB</td>
<td>71</td>
</tr>
<tr>
<td>4.6</td>
<td>Feasible design of LiveKB</td>
<td>73</td>
</tr>
<tr>
<td>4.7</td>
<td>OWL generated from TestRadio IDL interface</td>
<td>82</td>
</tr>
<tr>
<td>4.8</td>
<td>Representing an $n$-parameter IDL operation in OWL using the class pattern for $n$-ary relationships</td>
<td>84</td>
</tr>
<tr>
<td>4.9</td>
<td>Representing the get_gain_range operation from the OSSIE Radio_Control.idl in OWL using the class pattern for $n$-ary relationships.</td>
<td>85</td>
</tr>
<tr>
<td>4.10</td>
<td>Representing an $n$-parameter IDL operation in OWL using a list pattern for $n$-ary relationships.</td>
<td>85</td>
</tr>
<tr>
<td>4.11</td>
<td>Representing the get_gain_range operation from the OSSIE Radio_Control.idl in OWL using the list pattern for $n$-ary relationships.</td>
<td>86</td>
</tr>
<tr>
<td>4.12</td>
<td>Top-level classes with a subset of properties defined in the CRO ontology</td>
<td>88</td>
</tr>
<tr>
<td>4.13</td>
<td>Fragment of the Cognitive Radio Ontology.</td>
<td>90</td>
</tr>
<tr>
<td>4.14</td>
<td>A sequence of property assertions necessary to indicate the value of the carrier frequency</td>
<td>91</td>
</tr>
<tr>
<td>4.15</td>
<td>OWL ABox that results in ambiguous datatype property chains</td>
<td>93</td>
</tr>
<tr>
<td>4.16</td>
<td>RDF graph augmented with loops to disambiguate property chains</td>
<td>95</td>
</tr>
<tr>
<td>4.17</td>
<td>Object tree created by Invoker to access SDR’s object in runtime.</td>
<td>101</td>
</tr>
<tr>
<td>4.18</td>
<td>Representing ABox axioms as connected and disconnected RDF graph</td>
<td>103</td>
</tr>
</tbody>
</table>
4.19 Example of an ABox graph where multiple individuals of the same class are necessary

4.20 APIs involved in the CRF architecture

4.21 UML sequence diagram showing CRF’s initialization stage

4.22 Sequence diagram showing one scenario of handling incoming control message

4.23 Division of roles under CRF architecture

5.1 A screenshot of the GUI for the CRF implementation used during the SDR’10 Technical Conference demonstration

5.2 Application of LiveKB in self-controlling software

5.3 Screenshot of a Kalman Filter demo

5.4 Domain-API base architecture (left) and CRF (right)

A.1 Top-level view of the FIPA FSM

A.2 FIPA Query protocol

A.3 FIPA Request protocol

A.4 FIPA Contract Net protocol
Chapter 1

Introduction

1.1 Software Defined Radio

Software Defined Radio (SDR) – the term first introduced by Mitola [1] – is a radio in which some or all of the physical layer functions are *software defined*, i.e. radio’s operating functions are implemented with the use of software processing, as opposed to being implemented in hardware [2]. Eric Blossom, the author of the GNU Radio project, intuitively describes [3] the SDR as “the technique of getting code as close to the antenna as possible” and “turns radio hardware problems into software problems”.

The architecture of the SDR (Figure 1.1) comprises multi-band antennas, a radio frequency (RF) front end, a digital back end, which facilitates the software, and Digital to Analog (DAC) and Analog to Digital (ADC) converters, which bridge the RF with the digital components. SDR’s hardware may consist of only a few radio-specific components, but it at least includes multi-band antennas, DAC and ADC converters. The remaining functionality, such as the intermediate frequency (IF), baseband and bitstream processing, is usually implemented in a generic processing engine, which consists of general-purpose processors (GPP), digital signal processors (DSP), field-programmable gate arrays (FPGA) and other
computational resources, sufficient to include a wide range of modulation types [4]. The digital back end executes the software, which defines the radio’s physical layer functionality.

Figure 1.1: Software Defined Radio (SDR) architecture

The number of radio’s hardware components that can be reconfigured in software has been increasing since the SDR was introduced nearly two decades ago [1]. An ideal SDR architecture would consist of three units — a reconfigurable digital radio, a software tunable analog front-end and software tunable antenna systems. The analog front-end is limited only to components that must be implemented in hardware, such as RF filters, power amplifiers or data converters. However, due to the current limitations such as size, cost, power, performance, or processing times, an ideal architecture would be too costly. As a result, existing SDR architectures consist of various combinations of digital and analog components. Advances in SDR enabling technologies, such as DSPs and FPGAs, allow the functionality, once implemented only in hardware, to gradually shift to the software layer [5].

By shifting the burden of defining the radio from hardware to software, SDR offers a very high level of flexibility. It enables creating communication systems capable of multiple mode operation within a single hardware configuration [6]. It significantly reduces the cost
of the radio development because the software can be redeployed across different hardware, or reconfigured on the fly to provide a new feature to the user. Since the majority of SDR’s functionality is enabled in software, it allows for entirely new possibilities, inconceivable in traditional radios.

1.2 Cognitive Radio

Cognitive Radio (CR) is a term that was coined by Mitola in 1999 [7, 8, 9] to describe a radio communication paradigm which takes advantage of the SDR architecture and allows for dynamic changes of radio’s operational behavior in order to allow to achieve a variety of goals, including interoperability, performance optimization, opportunistic use of resources and others. In his work he recognized the need for the radio to have a computational description of its own structure, so that it would know what it knows, or in other words, be self-aware, as well as the description of the communications environment and goals. This knowledge could then be utilized by different radios to negotiate various aspects of their communication etiquette. Such knowledge exchanges could possibly result in an extended battery life or an increased payload data rate without the need to introduce new industry standards. To meet the description requirements of the CR paradigm, Mitola proposed to use a new language, the Radio Knowledge Representation Language (RKRL).

1.2.1 Cognition Cycle

When Mitola first introduced the concept of CR, he described a cognition cycle [7, 8], which was based on the Observe-Orient-Decide-Act (OODA) feedback loop [10], developed in the late 1970’s by US Air Force Colonel John Boyd to model real-time decision-making process. OODA (Figure 1.2) is a “cyclic model of four processes interacting with the environment, possessed by an agent that interacts competitively with other agents ” [11]:
- Observe – acquire information from the environment, receive guidance from the Orient process and feedback from Decide and Act processes.

- Orient – Also known as situation analysis, or situation assessment is “an interactive process of many-sided implicit cross-referencing projects, empathies, correlations and rejections” [10].

- Decide – Make a choice about the environmental situation and possible responses to it given the feed-forward from the Orient process.

- Act – Implement the chosen response given guidance from the Orient process and feed-forward from Decide.

![Diagram](image)

**Figure 1.2:** Observe-Orient-Decide-Act (OODA) feedback loop (Source: [10]).

Although OODA originated in the military domain, it is not restricted to military operations and can be applied in other domains to aid the decision-making process. Despite popularity, OODA has its shortcomings and several modifications or augmentations have been proposed [11].

Mitola adopted OODA and defined what is now called the ideal CR architecture (iCRA) [12], which adds two more explicit processes to the loop – Plan and Learn (Figure 1.3). The Plan process consists of planning decisions for the long term, and Learn supports machine learning based on past experience. A normal operation of iCR is characterized by the loop:
Observe-Orient-Plan-Decide-Learn-Act (OOPDLA). The Orient process, arguably the most complex phase in the context of CR [13], distinguishes two special types of events. These events shorten the cognition cycle in order to address situations that require faster than normal response [8]:

1. *Urgent*, e.g. loss of connectivity, shortens the cycle to OODA.

2. *Immediate*, e.g. the main battery has been removed, shortens the cycle to OOA.

![Figure 1.3: iCR – ideal cognition cycle designed by Mitola (Source: [7]).](image)

In the context of CR, the processes inherited from OODA gain more specific descriptions:

- **Observe** – collect observations about the environment using an array of sensors.
- **Orient** – integrate the observations with radio policies, user preferences and goals, hardware and software limitations and past experience.
- **Decide** – identify changes that need to be carried out in the SDR configuration in order to address a new situation.
- **Act** – reconfigure the SDR.

iCRA represents a very sophisticated vision of CR [12]:

5
(...). iCR continually observes (senses and perceives) the environment; orients itself; creates plans; makes decisions on its own and in conjunction with the user and external networks; and then acts. Actions may be physical, such as transmitting a signal, or virtual, such as associating a user’s action with the current situation.

As the name suggests, this is an ideal CR and it has not yet been fully implemented. Ten years after Mitola introduced the concept of CR, theoretical research was blooming with more than 30 special-issue scientific journals and over 60 dedicated conferences and workshops [14]. However, as the number of papers regarding CR increases every year, the number of demonstrations remains rather steady [14], which leads us to believe that there are difficulties with the practical implementation of Mitola’s vision. Researchers typically address challenges associated with each of the processes in the cognition loop.

1.2.2 Definition

The concept of CR drew a significant attention from different groups in the wireless communications community, however, the term has been applied as a solution to a wide range of problems in an inconsistent manner. To tackle this issue the Software Defined Radio Forum (now known as the Wireless Innovation Forum (WINNF)) provided nomenclature to define CR and its components [2]. According to [2], CR is defined as a radio designed according to the cognitive radio engineering paradigm that utilizes SDR, Adaptive Radio and other technologies; it is equipped with awareness, reason, and agency to intelligently alter its operational characteristics. The awareness requires that CR can perceive and retain information derived from its location, its environment, its internal state, node capabilities, and current needs of its user. CR is required to be endowed with the capability to reason over the perceived information by applying logic and analysis. Agency requires that CR can make and implement choices about its operational aspects based on the results of reasoning. Finally, CR is required to exhibit intelligent behavior by performing these choices in a
manner consistent with a purposeful goal.

1.2.3 Applications of CR

One of the most popular applications of CR is the opportunistic use of available spectrum, termed Dynamic Spectrum Access (DSA)[15]. DSA is a process of increasing spectrum efficiency through adjusting radio’s resources based on local spectrum sensing and establishment of cognitive radio networks [12]. DSA has been successfully demonstrated [16] and is now being integrated in some military and commercial products. However, Mitola’s vision of CR goes well beyond the DSA[12] and there are numerous use cases in which CR could be utilized. The use cases are traditionally divided into three areas of application [17], for instance:

- **Public Safety** – support for interoperability between different systems (police, ambulance, etc.), creation of rapid self-organizing networks in emergency situations.

- **Military** – support interoperability between heterogeneous military communication devices and systems, building ad-hoc and self-organizing networks in combat situation.

- **Commercial** – improvement in the quality of service, ad-hoc peer-to-peer networks.

1.3 Implementing CR

As of now, the wireless community has not developed a reference architecture for CR. Instead, different groups try to tackle this issue by proposing different approaches [18, 19, 20, 21, 22, 23, 24, 25, 26], each stressing different aspects of CR. In this research, we are proposing a new architecture for implementing CR. Figure 1.4 shows our vision of CR in relation to SDR. The main component of the presented architecture, Cognition Software (CS), is located outside of the SDR component. SDR consists of some software and hardware components. The
software part, typically delivered as firmware, allows application developers to access the SDR’s functionality. It is also responsible for interacting with the hardware components of the SDR using drivers. The operational behavior of SDR can be altered via its parameters. The observable parameters, traditionally referred to as meters or perceptions, allow the CS component to retrieve radio’s contextual information that can change over time. This information may come directly from the SDR itself, but also from additional sensors. CS takes advantage of that information and alters the radio’s operational behavior using its controllable parameters, usually referred to as knobs or actions.

Figure 1.4: Abstract view of a Cognitive Radio architecture

In relation to Mitola’s iCRA, meters allow for observation, CS is responsible for using that observation to orient and decide, and knobs are used to actuate decisions made by CS. Although CS may also perform planning and learning, we do not focus on this aspect of the cognition cycle in this work.
1.3.1 Knowledge Based Implementation of CS

The CS component can be implemented in many ways, employing different artificial intelligence or optimization algorithms. Existing CR architectures utilize different techniques such as genetic algorithms, case-based reasoning [27], knowledge-based ontological reasoning, or systems modeling complex intelligent behavior, to name a few. [13] provides an overview of existing CR architectures with the emphasis on the type of applied cognitive technology, and [28] provides a survey of applications of AI in the CR domain. The focus of this work is mostly on the knowledge based implementation.

This approach stems from the production systems architecture where an inference engine component, also known as reasoner, plays the central role. This approach separates the knowledge about the particular domain from the flow of control (algorithm). The control flow is handled by the reasoner that is oblivious to the domain, i.e. it is general-purpose.

The knowledge about the domain is formulated by domain experts and is encapsulated in the Knowledge Base (KB) in the form of ontology and rules. Ontology defines common terms and concepts used to describe and represent the domain knowledge. Rules, similarly as ontologies, have a declarative nature and express the logic of a domain-specific program without describing its control flow, which allows the reasoner to infer new facts given some initial facts. Rules may be augmented by the use of procedural attachments — commonly used functions implemented in imperative programming languages, e.g. arithmetic operations. Figure 1.5 shows how this paradigm fits in with the CR architecture: information that comes from the meters becomes part of the reasoner’s KB; reasoner executes the rules which in turn trigger changes in the SDR’s knobs.

Since the knowledge-based approach to the implementation of CR does not require the control structure of the software to be explicitly specified, it provides for a flexible framework for domain experts to design and alter CR’s behavior. They may not always be skilled software developers, familiar with the particular programming language used for the controller’s
hard-coded algorithm implementation. As contrasted with the fixed order of invocation, typical for the imperative programming languages, rules are not executed in a prescribed order, rather the system itself makes these decisions at the runtime [29]. By removing the burden of prescribing the invocation sequence from the developers, this paradigm allows different groups of experts to work on different policies (sets of rules) independently. Moreover, rules do not need to be compiled, which makes them very suitable for run-time deployment. Separating the policies from the device-specific implementation yields an additional benefit — the process of certification and accreditation becomes significantly simplified because each policy set needs to be accredited only once. Once accredited, the policies can be deployed on different radios without the need for additional certification, provided that the radio makes use of a certified reasoner that can interpret rules in the given language [30].
1.4 CR and self-controlling software

The knobs and meters in the CR architecture can be interpreted as part of a feedback loop, typical for dynamic systems. As it turns out [31], software not only can be used to implement controllers in dynamic systems, but it can itself be treated as the controlled system (in control theory called plant). Figure 1.6 shows how the problem of controlling SDR by CS can be abstracted using the framework for specifying and designing software that controls itself as it operates. SDR is treated as the plant, the control inputs, labeled $\alpha$, are the measurable inputs to the plant and correspond to the knobs of the CR. The observable outputs, labeled $\delta$, correspond to the meters in CR. The input from the environment, labeled $\Theta$, represents the part of the contextual information, some of which is sensed by SDR’s hardware, and some treated as disturbance. Controller, QoS (quality of service) and Goal constitute a component equivalent to CS. The goal, specified by domain experts, may refer to different aspects of radio’s operation, e.g. minimize energy consumption or maintain signal to noise ratio within a specified range. Controller is responsible for changing the values of the control inputs to the plant and the QoS component computes the feedback. The entire system tries to satisfy an externally defined goal (Goal). In the case of CR, Goal is specified by domain experts and most likely is not a part of the CS implementation itself. A goal may be defined by a set of rules.

In order to enable communication between the controller and the software-defined plant, one must know how to interface them together in a flexible fashion. It seems reasonable to expect that a solution to this problem would contribute to the general problem of interfacing any plant with a controller in a self-controlled software framework, regardless of its domain.
Figure 1.6: Cognitive Radio viewed as a special case of a control theory-based software model
Chapter 2

Problem Formulation

2.1 Introduction

Typically, in order to access the functionality of an external software component, developers interact with it via its Application Programming Interface (API). An API is a set of calls that may be invoked by external entities, such as other components or layers. An API is documented by the developer of the particular component. The documentation specifies the way in which others may access the services through the API. Most of proprietary software is closed-source and an API is the only way to interact with it. In such situations the API may be a measure to ensure proper usage of that component. A good example of this paradigm are APIs provided by an operating system, which provide access to the underlying hardware resources, e.g. video graphics API.

A thorough overview [32] of the link layer software interfaces offered by different wireless communications standards conducted in 2004 shows that numerous APIs can significantly differ from each other, because they are designed for specific technologies, e.g. Bluetooth, WLAN or GSM. Although several years have passed, the wireless communications community still has not established and adopted standard APIs for controlling software radios.
As a matter of fact, most of current PHY, LINK and MAC interfaces are proprietary and hidden from the developers, which causes serious difficulties for the CR development [33]. Numerous APIs that exist today are provided by device vendors in a technology dependent manner, e.g. 802.11 MIB [33]. This situation leads to the main problem with implementing CR according to the knowledge based approach described in the previous section: lack of a standard CR architecture and a standard API for interfacing reasoners with the SDR’s knobs and meters.

### 2.2 Requirements

We identified the requirements for such an architecture and a standard API as enumerated below (see Figure 2.1):

![Diagram showing requirements](image)

**Figure 2.1:** Visual representation of the problem formulation

a) **Reusable knowledge:** The CR architecture should support the reuse of ontologies and rules written by domain experts. In other words, it should be possible to use rules written by domain experts, rather than programmers and rules should capture the generic domain knowledge, rather than the knowledge of a given radio implementation. Moreover, radios should be able to exchange knowledge with each other.
b) **SDR-independent:** The CR API should not depend on a specific SDR interface. In other words, the API should allow for interfacing the knobs and meters of any SDR.

c) **Platform-independent:** The CR API should be hardware/software platform independent. In other words, the architecture should allow for interfacing knobs and meters of SDRs implemented on various platforms, i.e., implementations in different languages on different operating systems and hardware.

d) **Domain-adaptable:** The CR API should allow for relatively easy adaptation to changes in the domain. For the SDR domain this means adaptation to new SDR concepts and technology. The architecture should be flexible enough to be able to adapt to such changes without the need to recode, recompile and redeploy.

### 2.3 Literature Review

SDR software can be looked at as a black box component for which vendors provide an API which enables access to its knobs and meters. As mentioned earlier, the big challenge of designing an interface with a cognitive engine is the fact that different radios provide different APIs. However, since this is quite a common problem in software engineering, some generic solutions already exist to tackle this issue. In this section we examine how these solutions apply to the radio domain by reviewing a number of existing CR architectures. Since the problem of interfacing radio components is independent of how the cognitive software is implemented, the review also includes CR architectures that do not follow the knowledge-based approach. We categorize related work based on the type of interface that is employed to access and control radio components.
2.3.1 SDR API

The straightforward solution to interact with different SDR platforms is to implement an intermediate dedicated layer for each one of them (see Figure 2.2). Using an appropriate API layer, cognitive engine directly invokes methods of the given SDR’s API in order to get the current state of the SDR’s parameters. The process in which the SDR-specific API is translated into the engine’s internal calls can be proprietary (the choice of the supported platforms can be a business-oriented decision).

![SDR API Diagram](image)

**Figure 2.2:** The SDR API approach to interfacing various radio platforms

The SDR API design supports only those SDR platforms for which a dedicated piece of software was implemented by the cognitive software provider. The cognitive engine has no means to interface with unsupported platforms (SDR C in the picture). Furthermore, the SDR developers may have no access to the cognitive software’s internal structure which prevents them from implementing the interface themselves. Most likely, this type of CR would be delivered in different versions, each for a different SDR.

Cognitive Radios that establish the SDR type of API, fall into one of the following categories:

1. Commercial products with proprietary interfaces, for instance:
(a) xMax\textsuperscript{1} infrastructure from xG Technology, Inc., which implements one of the CR’s most popular applications – dynamic spectrum access – in the cellular space. The infrastructure comprises base station, mobile handset, mobile switching center and network management tools.

(b) XG1\textsuperscript{2} radio from Adapt4, which operates in the 217-220 MHz band and provides cognitive functionality, such as dynamic frequency selection and interference avoidance.

These proprietary technologies are closely tied to very specific radio platforms, which prevents other SDR vendors from extending the support to their products.

2. Research projects focused on experimenting with the CR on a particular radio platform of choice, e.g. Cognitive, Radio-Aware, Low-Cost (CORAL) research platform \cite{34} developed by Communications Research Centre (CRC) in Canada. This platform is based on a ”shell” built around the IEEE 802.11 standard and provides a Cognitive Network Management System (CNMS). Researchers can use CNMS and a set of provided APIs to explore various ideas related to cognitive radio networks.

Summary

Applying the SDR API approach to the wireless domain leads to several major problems:

- The reasoner is able to communicate with only as many SDRs as the number of implementations that are provided with the cognitive software. It is limited to the SDR platforms that are known and supported at the design time.
- This approach is also platform-dependent since the reasoner would have to have special code not only for specific SDRs, but also for all implementations of the same SDR

\textsuperscript{1}xMax website is located at \url{http://www.xgtechnology.com}
\textsuperscript{2}Adapt4’s product information is available at \url{http://www.adapt4.com/}
on different platforms (implementation in different languages, running on different operating systems or hardware platforms).

- Any update to a supported SDR-specific API that is not backwards compatible with its previous version requires an update of the part of code in the cognitive software that is responsible for the interface with that SDR. Therefore, in order to continue support for this platform, it needs to be recoded, recompiled and redeployed. Even if the update is backwards compatible, it would still require the same process in order to take advantage of possible technological advances (given that the cognitive software supports it).

Clearly, SDR API does not satisfy our requirements — out of the four of them only one (reusability of knowledge) is partially supported (see Table 2.1). The reason for that is the fact that this approach is primarily focused on providing cognitive capabilities to a specific platform, rather than making it available to an array of different platforms, which is the focus of this work. Such a design limits the portability of the cognitive software and the possibility of establishing interoperability between heterogenous wireless devices — a feature especially desired in the military domain [35].

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reusable knowledge</td>
<td>Partial</td>
<td>Depending on the actual implementation it may provide reusable rules and ontologies, but only to a limited number of supported SDRs.</td>
</tr>
<tr>
<td>SDR-Independent</td>
<td>None</td>
<td>It requires a SDR-specific piece of code for each supported radio.</td>
</tr>
<tr>
<td>Platform-Independent</td>
<td>None</td>
<td>It requires special software for every implementation of an SDR on every platform. The focus is to interface with a single radio platform at a time.</td>
</tr>
<tr>
<td>Domain-adaptable</td>
<td>None</td>
<td>Changes to the SDR API must be followed by an update to the interface code, then the code must be recompiled, tested and redeployed.</td>
</tr>
</tbody>
</table>
2.3.2 Standard API

The Standard API approach (illustrated in Figure 2.3) is somewhat similar to the previous one, although it allows for a less restrictive design in which the communication between a cognitive engine and a SDR platform is possible due to the implementation of an open standard by both parties. It is based on the idea that there is a Standard API which must be implemented by every SDR platform to be supported by the framework. SDR developers implement the standard and allow the reasoner to access the radio’s functionality by invoking methods defined in that standard (SDR A and B in the picture). This is a very common practice for solving interface problems (not only in software engineering), nevertheless, it does not come without drawbacks.

![Figure 2.3: The Standard API approach to interfacing various radio platforms](image)

The main advantage of this approach for the cognitive software developers is that there is no need for them to have design-time knowledge about particular SDR implementations. However, the Standard API approach does not support the SDR platforms which provide an API that is not compatible with the standard, e.g. the legacy systems developed before the standard was published. This problem could be alleviated by applying the Adapter [36] design pattern, i.e. a generic solution to common problems in software design. For instance,
the adapter could be used to wrap each non-standard SDR API with a dedicated piece of software in order to make it compatible with the cognitive engine by delegating the standard API calls to the SDR-specific methods (SDR C in the picture). Thus, by using this design pattern, with some additional work, it is possible to extend the SDRs that do not support the standard API so that they can interface the reasoner - a task that they were not originally designed for.

Having a standard API, SDR can communicate with the cognitive engine either in a pull or a push fashion. In the first case, SDR accesses the engines’s functionality on demand and in the latter the engine invokes some callback methods in the SDR. This paradigm could be implemented using the Plugin [37] design pattern — in the case of the pull communication, the cognitive engine becomes a plug-in for the SDR, while in the push pattern the roles are reversed. The big advantage of taking this approach is that the standard API does not assume existence of any particular SDR or a particular engine. Any reasoner and SDR platform that properly implement the standard may be interfaced with each other. Usually, the consumer of the plugin is set up to use a selected implementation based on some control structure, configuration file, or by placing the plugin in a specified location in the file system.

A special case of the Standard API approach is one in which the standard is defined by an eXtensible Markup Language (XML) document (Figure 2.4). We refer here to this type of interface as an XML-API. Rather than directly invoking API methods, software components using XML-API exchange XML documents over some transport protocol. When documents are received, they need to be first parsed and then processed. Conversely, before a response is sent back, it needs to be serialized into an XML document. The exchange of messages is possible because its participants produce and expect the same type of messages with a well-defined format. The standard is usually accompanied by Document Type Definition (DTD) or XML Schema, which are used for document validation.

XML-API introduces a better support for heterogeneity, because components must only
Figure 2.4: Using XML-API approach to interface cognitive engine and various radio platforms use the same transport layer (e.g. TCP, UDP or simple sockets) and the same message exchange protocol (e.g. SOAP), and these are well standardized and available on numerous software and hardware platforms. However, in order to use XML as the interface language, additional layers need to be present in the system: XML parser/serializer and implementation of a transport and message exchange protocols. The latter two are domain independent with support from most programming languages via ready-to-use libraries. However, the parser/serializer is the piece of code that needs to be written and maintained for each software platform and for each XML standard document. In this sense, XML-API is no different than the Standard API approach.

It is important to make a note that using XML-API with an inference engine requires a dedicated conversion of the XML documents to an ontological format. This conversion cannot be automated because XML has no semantics [38, 39]. Although XML was designed with human and computer consumption in mind, what may seem intuitive to a human reader
is not intuitive to a software agent, unless the data is semantically annotated, as in Resource Description Framework (RDF) [40].

The success of any standard, XML-based or not, heavily depends on its popularity and unless there is a critical mass of users, it is of low value. Unfortunately, there is no standard API to interface the cognitive software with different SDRs, although there are efforts under way to establish new standards. Some of them regard specific parts of the CR domain, e.g. DSA API [41]. Several open CR architectures that have been proposed, albeit not by any standards body, could possibly serve one day as a standard. Among them are the Cognitive Engine [42, 24] and VTCROSS Cognitive Radio API [25] from Virginia Tech, and the Community-Based Cognitive Radio Architecture (CBCRA) [21, 22]. Description of these architectures and their APIs follows next.

**Cognitive Engine from Virginia Tech (VTCE)**

VTCE architecture was designed to support the use of genetic algorithms [43, 44] in the design of cognition software in CR. Genetic algorithms are entirely generic and can be applied in various domains. These algorithms to a certain degree mimic the natural evolution process in order to search for an optimal solution. VTCE aims to optimize radio’s operation by searching for optimal communications parameters.

The main component of VTCE, Cognitive Engine, communicates with SDR platforms using an XML-based API. Creating a waveform on a particular radio using the VTCE architecture requires providing XML and DTD documents for each of the following:

1. Description of the waveform as a collection of knob descriptions. Each knob needs to be described by its range of values using the minimum, maximum and a step size values. Using the genetic algorithms terminology, this is a description of a *chromosome*.

2. Description of objective functions, which allow to pick the *fittest* solution (assignment
of knob values).

3. Description of meters and sensors, which are required to compute the objective function values.

Explanation of the interface between the radio and the cognitive engine in VTCE requires some understanding of how the genetic algorithm operates. The descriptions of knobs and meters have no semantic meaning, the algorithm merely tries to find the optimal values for all knobs given the objective functions that are calculated based on the data from the meters (note that the algorithm may often generate completely useless waveforms). At startup, a random assignment of knob values is generated and the objective functions are calculated. As the genetic algorithm progresses, it chooses new values for the knobs within their described range, each time increasing or decreasing their value by a multiple of the respective step size. Every new set of knob values, termed generation, is parsed and a new waveform is created. The cycle of creating new waveforms continues until a termination criterion is met, for instance, the total number of generations reached a certain threshold, or a specific amount of time has elapsed.

Rondeau explains [42] how the genetic algorithm can also be enhanced by Case-Based Decision Theory (CBDT), which incorporates historical knowledge into decision making within an optimization process. A state of the observed world, described in terms of values coming from the meters and sensors can be looked at as a single case. A system using CBDT consults previous cases and solutions (generated waveforms) applied to them. Given a similarity and utility functions, similar cases are fed into the genetic algorithm in order to aid the decision process. Moreover, each new case, along with its generated waveform and utility, is archived for future iterations. The system is further optimized by purging cases that no longer satisfy some criteria, for instance cases with low utility or low similarity values, or the oldest cases in the base.
The architecture of VTCE is entirely designed around the idea of the genetic algorithm. The algorithm is domain-independent and it does not require the meaning of meters and knobs to operate. The cognitive engine collects information about the SDR in the form of XML documents constrained by a corresponding DTD definition. The DTD lists possible names of knobs, meters and objectives, however, as with any plain XML document, these names have no semantics. The XML specification of the interface between the cognitive engine and radio platforms must be provided in a DTD document at design time. Because the means by which the data is actually collected from the radio and serialized into XML are outside the scope of the architecture, this design results in requiring to write a dedicated piece of code for each radio.

Furthermore, each of the SDR’s components must implement a simple state machine, which consists of 4 states: 1) Initialization, 2) Waiting for data request, 3) Collecting and serializing data and 4) Transferring data. Using this automaton, CE can poll radio’s components to get their XML descriptions, although the transfer protocol upon which the data exchange occurs, is specified not by the architecture.

Although CE does not directly invoke the SDR’s API methods and the DTD documents are not proprietary, the framework requires radio-specific layers, which need to be maintained and recoded if the radio API changes. In addition, if the DTD gets replaced or is augmented, all parsers need to be updated as well.

Since the VTCE architecture was mainly designed to support the genetic algorithm, its use is rather limited to use cases where optimization is the key objective. As such, there is no support for knowledge reusability or exchange between different radios. Rondeau [42] mentions how CR networks could be utilized to collaboratively execute distributed genetic algorithms, but there is no mention of a language that the radios could use to communicate with each other. Thus, use cases such as the network extension [45] could not be executed on this platform.
VTCROSS

Virginia Tech Cognitive Radio Open Source System (VTCROSS) framework is an extension of the VTCE architecture (described above). The project was designed around a strong belief that the CR architecture should be modular and consist of well-designed and strict APIs. The main component of the system is a Cognitive Radio Shell (CRS), which is configured via an external XML file. In a similar fashion to VTCE, the XML configuration lists available parameters with their minimum, maxim and step size values, providing enough information for the genetic algorithm to operate.

CRS communicates with other components in the system using the following APIs (available only in C/C++):

- **Cognitive Radio API** – this API, responsible for accessing and modifying radio’s operational parameters, is of our particular interest and is described below.

- Cognitive Engine API – allows different types of engine to be connected and provide cognitive capabilities to the SDR. Although the authors aim to support any type of cognitive engine, in their work they only mention those where the cognitive logic is maintained in the software itself, on the implementation level, usually based on case-based reasoning and genetic algorithms [18].

- Policy Engine API – allows use of global policy engines to validate waveforms generated by the cognitive engines. This component was inspired by the Policy Conformance Reasoner from the XG platform [26], described in more detail below.

- Management Service Layer API – is an API to control an array of cognitive engines, which can collaborate to achieve a goal.

Within the Cognitive Radio API the following three methods are key for accessing radio’s parameters:
The first two methods correspond to retrieving values of available meters and knobs, respectively. The third method returns a list of optimal values for parameters calculated by the cognitive engine. Although these methods are generic, they were designed with the genetic algorithm in mind and could not be easily used in conjunction with an inference engine. Representing these values in the knowledge base would require having a design-time knowledge of the XML configuration file passed to the CRS in order to translate them to a particular ontology. This stems from the fact that XML has no semantics and cannot be automatically consumed by a reasoner (as explained above).

A dedicated method in the API for calculating optimal parameters indicates the focus of this architecture on the optimization use cases. In addition, while VTCROSS heavily relies on the use of APIs, none of them is an accepted industry standard.

Community-Based Cognitive Radio Architecture (CBCRA)

The focus of the CBCRA architecture described by Ginsberg et al. [21, 22] is to take advantage of the Semantic Web technologies to enable Dynamic Access Spectrum (DSA) and to enhance system development and interoperability. Ginsberg et al. bring attention to the fact that the SDR is implemented in object-oriented languages with imperative semantics in nature. On the other hand, the languages used for representing knowledge and policies are usually declarative. The proposed architecture aims to bridge the two styles of development allowing experts of each one to work independently.

Perception and Action Abstraction Layer (PAAL), part of the CBCRA, is a SDR-independent abstraction layer that separates cognitive applications from particular SDR
platforms. This layer is essentially an API and it is assumed that SDR interfaces will be wrapped to match methods defined in its terms. In order to achieve independence from a particular cognitive engine, an Ontology and Rule Abstraction Layer (ORAL) is defined. This layer is envisioned to be based on Web Ontology Language (OWL) [46] and Rule Interchange Format (RIF) [47], standards developed by the Semantic Web community in order to represent ontological and rule-based knowledge, respectively. Although no implementation details are provided, it seems that ORAL is also envisioned as an API. The cognitive application software interfaces the SDR via PAAL, and the cognitive engine via ORAL. The CBCRA framework aims at the separation of concerns between the SDR developers and the knowledge engineers by defining appropriate layers. Unfortunately, similar to the VTCROSS architecture, it is an API-centric approach without the industry support for its APIs.

Platform Independence

A feature that is tightly coupled with the standard API is platform dependence. In VTCE this challenge it tackled by use of an XML-API with a support of standard transport protocols. Within the VTCROSS architecture all components must implement a custom interface and communicate with each other with the use of socket connections over a TCP/IP network [25]. While socket programming might be straightforward and does not require much protocol overhead, it is very low-level and has no support for complex data types or error checking. Moreover, sending binary data over the sockets is limited to the machines that use the same binary format. Perhaps because of these issues, VTCROSS components exchange pure ASCII messages with a plan to introduce a custom protocol in the future. Socked-based communication is thus only to a certain degree platform-independent and it requires a fair amount of additional programming.

The authors of the CBCRA architecture embrace the platform-independence by building abstract layers on top of platform-independent languages and standards. The ORAL layer
is envisioned to be built on top of OWL and RIF, however, no analogous directives are given for implementing the PAAL layer. The described prototype implementation used Java-based mockups for the wireless devices, while the PAAL layer was implemented in the same language, thus entirely avoiding the issue of platform independence.

Summary

XML-API from VTCE, set of APIs from VTCROSS and PAAL are all domain-specific interfaces, i.e. they consist of methods that are radio domain-specific. Designing architectures that heavily rely on standard domain-specific APIs leads to major challenges, even if the community widely adopted the standard:

- The API may become a bottleneck of the design when the interface between two software components is frozen in order to support compatibility, despite the need for a major overhaul of the API.
- Creating a new version that is not backwards-compatible results in losing the interface among the components that implement different versions of the standard.
- Keeping up to date with API changes requires redesign, recompilation and redeployment of the part of code responsible for implementing the API. This requirement may not be suitable for portable devices.
- Non-compliant software needs to be wrapped with adapters which require constant maintenance in order to keep up with the changes made both in the standard and in the SDR-specific API.

Despite the fact that all of the architectures described in this section offer some level of flexibility in which the CR can support a number of SDR platforms, they fail to fully satisfy our requirements (see Table 2.2). Although the Standard API is a lot more flexible than the SDR API approach due to the introduction of layers which promote separation of
concerns, it is still vulnerable to API changes, i.e. significant development efforts are required to keep up with the changes made to the standard. As mentioned in the introduction to this section, the success of any Standard API depends on its support from the community. However, the introduction of a new standard and the maintenance of the existing standards typically requires a lengthy process which consists of several steps, such as creating and organizing work groups, a validation process, gaining tool support and community adoption. The inertia of updating standards is discouraging for fast-changing domains, such as wireless communications, where outdated and fixed APIs may prevent from taking full advantage of technological advancements.

Table 2.2 The Standard API approach and its support for the desired features of the CR architecture

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reusable knowledge</td>
<td>Partial</td>
<td>As long as the cognitive software and the SDRs implement the same standard API, rules and ontologies can be reused across different SDRs without modifications.</td>
</tr>
<tr>
<td>SDR-Independent</td>
<td>Partial</td>
<td>It requires that each SDR implements the standard API, either directly or by an adapter.</td>
</tr>
<tr>
<td>Platform-Independent</td>
<td>Partial</td>
<td>Although XML-API offers platform independence via use of standard transport protocols, platform-independence is not an inherent feature of this type of interface, cf. VTCROSS.</td>
</tr>
<tr>
<td>Domain-adaptable</td>
<td>None</td>
<td>Changes to the standard API must be reflected in the implementation of both the cognitive software and the supporting SDR platforms.</td>
</tr>
</tbody>
</table>

2.3.3 SCA-based interface

Unlike in the case of the architectures described in the previous chapter, which still lack industry support, SDR community has made use of well-defined standards, primarily the Software Communications Architecture (SCA) [6], which has been widely supported by both industry and the academia. Due to the SCA popularity it is crucial to identify means necessary to implement a CR architecture on top of SCA-complaint radio systems.
The SCA is a software architecture framework that was originally developed by the US Department of Defense for its Joint Tactical Radio System (JTRS) to support portability of waveforms across heterogeneous and distributed hardware. It provides a hardware abstraction layer and means to identify available computational resources of the radio system to match application requirements for those resources [4]. The SCA itself does not define an architecture for a software radio, rather it defines an architecture for the core management, deployment, configuration, and control of hardware and applications, which could pertain to virtually any domain [48]. When applied to the SDR, SCA provides the software infrastructure to develop waveform software on top of a radio system.

Figure 2.5 shows how the SCA can support the development of portable waveform applications. The SCA consists of a set of APIs which define how distributed components can communicate with each other by the means of ports. One could develop an SDR on top of the SCA by defining specialized components, each for a different layer of the network stack. Those components provide the API for the Physical (PHY), Media Access Control (MAC) and Logical Link Control (LLC) layers, security features and the input/output (I/O) of radio devices. At the heart of the SCA is Core Framework (CF), a layer that manages all software and hardware components on top of the Common Object Request Broker Architecture (CORBA), the platform-independent middleware. Waveforms are defined independently of the underlying hardware, allowing them to be ported across different platforms, ideally without any modification, although that is rather unrealistic due to differences between different board manufacturers [48].

Waveform applications in the SCA are executed in the Operating Environment (OE), which consists of CF, CORBA middleware and a POSIX compliant operating system [50]. Both software and hardware components are accompanied by metadata stored in XML files, which describe their interfaces (ports). Likewise, applications are described in a set of XML files, which specify their software and hardware requirements. All components and applica-
Figure 2.5: The Software Communications Architecture (SCA) (Figure adapted from [49] and [48], omits the non-CORBA components and their respective adapters.)

Components in SCA implement well-defined APIs which together with the XML descriptions allow CF to manage their lifetime, i.e. it can install, create, start and stop them on demand. Although SCA expects the components to be CORBA-capable, it also provides means to support legacy software that is not CORBA-capable. This approach to deploy and manage software was standardized outside of SCA by OMG [51] and can be utilized to build component-based distributed applications in virtually any domain.

Currently, two open-source implementations of the OE are available — SCARI–Open\(^3\) and OSSIE\(^4\) (Open Source SCA Implementation::Embedded). SCARI, developed by Communications Research Centre Canada and implemented in Java, was the first reference implementation of the SCA. OSSIE, developed by the Wireless@Virginia Tech group and im-

\(^3\)Available at [http://www.crc.gc.ca/rars/](http://www.crc.gc.ca/rars/)
\(^4\)Available at [http://ossie.wireless.vt.edu/](http://ossie.wireless.vt.edu/)
mented in C++, is aimed at supporting electrical engineers who may not be familiar with Java [52]. OSSIE provides an IDE with a set of additional interfaces to facilitate the development of waveforms.

Within OSSIE, radio components provide API that allows access to their operational parameters. Typically, each parameter can be accessed and updated with three methods of the form:

1. `void get_knob_range(in unsigned long channel, out float min, out float max, out float step)`

2. `void get_knob(in unsigned long channel, out float currentValue)`

3. `void set_knob(in unsigned long channel, in float newValue)`

Note that the first method simply provides information about the value range. It seems that this API was designed with the genetic algorithm in mind, for which this kind of information is required to operate. Most importantly, these methods are domain-specific (names of parameters appear in the names of the methods) and such API suffers from all the problems that are associated with the Standard API approach.

The SCA design provides a great deal of independence for the waveform developers. Not only waveform applications once written can be theoretically ported to any SCA-compliant hardware, but the developers can also develop software in many programming languages, as long as the languages support CORBA. Unfortunately, such flexibility comes at a cost – SCA requires waveform developers to provide and manage a large number of XML description files, sometimes exceeding hundreds [53], which may be virtually impossible without an IDE support and proper training. Furthermore, a complex process of validation is necessary in order to achieve SCA compliance.

The focus of the SCA is the management of distributed components, which is very important in the radio domain, because it enables the portability of waveform applications across heterogenous hardware. As such, it is of a great value for developing SDRs. However, the
SCA does not provide any support for the CR out of the box — there is no dedicated API for the cognitive software, thus there is no standard–supported way to realize the CR architecture on top of the SCA-based SDRs. Instead, one must treat every SCA-compliant SDR as a separate SDR, with its own specific API. One solution to this problem is to include cognitive capabilities in the application layer along with the waveform software. However, such a solution would be most likely proprietary and constrained by the existing APIs, available to the waveform developers, probably too generic to be used for this purpose [54]. Wellington suggests [54] that the cognitive software be implemented as an SCA-compliant component and interact with the waveform components by the means of a set of new interfaces.

OSCR

Open Source Cognitive Radio (OSCR) architecture [20] is an attempt to incorporate SCA-based radios into the design of a CR architecture. It tackles the challenge of interfacing a cognitive engine with SCA-compliant radios by defining a custom API, required to be implemented by SCA components which comprise the SDR (see Figure 2.6). OSCR Radio Interface (ORI) extends the SDRs with an interface that allows them to be configured and monitored by a cognitive engine. Essentially, the ORI works as an adapter and is required to be customized for each SDR in order to translate between the OSCR API and the SDR-specific API. The OSCR Radio Multiplexer (ORM) communicates with the ORIs using the CORBA middleware, but it provides only a C++ control interface for the cognitive engine.

The goal of the OSCR architecture — enabling the reuse of a cognitive engine across different SCA-compliant radios — is achieved at a significant expense:

- To be compatible with the architecture, each SDR requires a custom adapter, which requires maintenance.
- The architecture requires that the application layer must use sockets to transfer data (platform dependence).
The architecture breaks the platform independence provided by the SCA framework by offering only a C++ interface for the cognitive engine.

- The OSCR API is limited and comprises specific methods, e.g. `getModulator`, `getCoder` that are not standard, thus creating the risk of being modified in the future.
- The architecture does not support SDR platforms that are not SCA-compliant.

Summary

From the above discussion it is clear that the SCA itself does not solve the problem of realizing a universal cognitive radio architecture. In fact, the SCA has not been designed specifically for the radio domain. However, the problems mentioned above would persist even if the current set of APIs provided by the SCA was augmented by a new interface dedicated to the cognitive engine functionality. In fact, the SCA–based CR architecture
would resemble that of the aforementioned Standard API, facing the same issues described above. While this design would be superior to the one of the Standard API, because it would have the industry support behind it, it would only add the platform-independence support to the list of our requirements (see Table 2.3). In addition, it would make major changes to the API even more expensive, because not only they would require updating the implementation, but also a new SCA compliance certification.

Table 2.3 Features of the SCA-based design and the requirements for the CR architecture (given that a new API for the CR was added)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reusable knowledge</td>
<td>Partial</td>
<td>SCA supports software and hardware independence allowing the rules and ontologies to be portable, but only across the SCA-compliant SDRs.</td>
</tr>
<tr>
<td>SDR-Independent</td>
<td>Partial</td>
<td>It only supports SCA-compliant SDRs, which makes it even more restrictive than the Standard API.</td>
</tr>
<tr>
<td>Platform-Independent</td>
<td>Full</td>
<td>The need for software and hardware independence was one of the main forces behind creating the SCA.</td>
</tr>
<tr>
<td>Domain-adaptable</td>
<td>None</td>
<td>Changes to any of the SCA APIs must be reflected in the implementation of both the reasoner and the supporting SDR platforms, possibly followed by the SCA compliance certification.</td>
</tr>
</tbody>
</table>

2.3.4 Reflection-based interface

In all of the architectures described above the interface to the SDR was known at the design time and this knowledge was used in the design. Regardless of whether it was a custom SDR-specific or a standard API, in each case the cognitive software interfaced the SDR in a static fashion, i.e. the interaction process was hard-coded in the API. Should the API change and not be backwards compatible with the previous version, the interface would no longer be supported. Thus the CR–SDR interface in these architectures is not domain-adaptable.

Instead of utilizing some form of a static API, software components can interact with each other with the use of the language mechanism called reflection[55]. Reflection allows programs to observe and modify their own structure and behavior dynamically, at the run-
time. Listings 2.1 and 2.2 show the difference between invoking an object’s method using the standard invocation and using the reflection mechanism in Java, respectively\(^5\). In the traditional approach the name of the class and of the method must be known at the time of implementation, they are used directly in the code. The reflection mechanism allows for a more flexible design, because the name of the class and of the method can be provided dynamically, allowing the actual implementation to be independent of these names. Although using reflection results in additional lines of code for a single invocation, it becomes cost-effective when invoking a series of methods.

**Listing 2.1** Standard method invocation

```java
import mypkg.SDR;

SDR sdr = new SDR();
Long id = sdr.getID();
```

**Listing 2.2** Method invocation using the Java reflection

```java
String clsName = "mypkg.SDR";
String methodName = "getID";
Object result;

Class<?> cls = Class.forName(clsName);
Object obj = cls.newInstance();

for (Method m : cls.getDeclaredMethods()){
    if (m.getName().equals(methodName))
        result = m.invoke(obj, (Object[]) null);
}

Long id = (Long) result;
```

\(^5\)For the purpose of clarity, exception handling is not included in the source code.
Ontology-Based Radio (OBR)

OBR [56, 23] is an architecture, in which the reflection mechanism is used to access radio’s knobs and meters. The premise of the OBR is that the radio analyzes its own structure to gain self-awareness, rather than being constrained to the design-time knowledge about itself. Reflection allows the method names pertaining to a particular radio API to be fed dynamically, from within the rules. The key benefit of this approach is that the SDR-specific interface information is not part of the design, which allows the OBR to be flexible enough to work with different APIs, as long as the necessary information about them is dynamically provided.

The OBR architecture (Figure 2.7) was designed with the inter-radio communication in mind, where radios exchange knowledge among them in order to achieve a variety of goals defined by their policies. The architecture distinguishes two types of messages exchanged between radios: data and control. The first type stands for the regular data sent and received directly by the application layer. The control messages include the knowledge that is consumed and produced by Reasoning Component (RC). Incoming data and control messages are dispatched by the Data In component to the Data Sink and the Monitor Service (MS) components, respectively. MS forwards control messages to RC, but also provides it with the SDR’s contextual information, extracted from the SDR software using the reflection mechanism. As a result of inference, Reasoning Component produces new control messages that are sent to other radios, but also requests updates to its own SDR’s parameters. The updates are implemented by MS using the reflection mechanism. The generated messages, along with the data from the application layer, are forwarded by the Data Out component to the SDR’s transmit path.

By the means of reflection and the ontology-based reasoning, radios can identify their contextual information and infer the communication protocol being used. They exchange knowledge to negotiate parameters of their communication. It has been shown that this
architecture can be used to analyze the multipath structure [56], negotiate the length and structure of the equalizer training sequences [23], and dynamically extend network coverage and reachback [45].

Even though the OBR architecture presents a great potential for a flexible interface between a reasoner and the SDR, up to this point only a limited prototype has been implemented, using both Java and C. A program that allowed for the access to the sound card simulated the physical layer; it was written in C. The Data Link (DL) layer was implemented in Java and communicated with the C program using the Java Native Interface (JNI) framework. Finally the MS, also implemented in Java, was interacting with the DL using the Java reflection. The implementation of the OBR prototype was possible due to some significant assumptions, in particular, the SDR’s API was required to be implemented in Java or at least accessible via JNI. Moreover, the names of the parameters used in the rules directly corresponded to the names of the objects and methods in the Java code, which means that the rules would have to be adjusted for any SDR platform different than the one used in the experiment. Additionally, the exchange of messages between the MS and the DL layer was implemented using an API that was known at the design time.
The OBR architecture does not fully support any of the requirements (see Table 2.4), which can be explained by the fact that the OBR was developed mainly to show how the SDR could take advantage of self-awareness and exchanging knowledge between other radios. Despite its novel approach to interfacing with the SDR using the Java reflection, the OBR is platform-dependent. Since it is ontology-based, it supports writing rules that use terms from the ontology. It is the only architecture which provides a somewhat flexible interface where API method names are maintained at the rules level, rather than used in the hard-coded invocations. Thus, the SDR-specific interface information is not part of the design.

Table 2.4 OBR architecture and its support for the desired features of the CR architecture

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reusable knowledge</td>
<td>Partial</td>
<td>Rules correspond directly to the underlying SDR implementation.</td>
</tr>
<tr>
<td>SDR-Independent</td>
<td>Partial</td>
<td>Although the SDR-specific information is not included in the implementation of the interface, it is included in the rules.</td>
</tr>
<tr>
<td>Platform-Independent</td>
<td>Partial</td>
<td>The reasoner may interface with different SDRs by the means of JNI, which allows for executing native applications. However, JNI requires special implementation and supports only C and C++ languages.</td>
</tr>
<tr>
<td>Domain-adaptable</td>
<td>Partial</td>
<td>Updates to the SDR API should only require changes within the rules, which are used to invoke the reflection mechanism. Having said that, the implementation of this mechanism was to a certain degree dependent on the underlying SDR platform.</td>
</tr>
</tbody>
</table>

2.3.5 Unspecified interface

There are several CR projects that do not specify the radio interface details at all and instead, focus on specific use cases where CR can be applied, using policies and abstract descriptions. A good example for this category is the NeXt Generation Communications (XG) Program from Defense Advanced Research Projects Agency (DARPA) [26], described below.
DARPA XG

Having support from DARPA, XG is one of the most advanced projects within the CR domain. Its main focus is the use of DSA to operate on unused spectrum without causing interference to incumbent users that are not “XG-enabled” [13]. According to [15], its goal is to develop a de facto standard for CR and dynamic spectrum regulation. One of the key principles behind XG – independence from radio architectures – was achieved by not defining implementation standards, leaving them to implementors. Instead, XG describes behaviors using a policy language, abstract from specific radio architectures. Policies specify what a radio system should do when changes in the environment occur, while ensuring compliance with regulations. A good explanation of the relationship between the XG policies and the actual protocol implementations is given by [57]:

It can be seen that the policy is an abstraction of the actual protocol: the policy describes what has to be done, but not how to do it, which is performed by the protocol.

Figure 2.8 shows an abstract view of an XG-enabled device. Its two major components, Policy Conformance Reasoner (PCR) and a System Strategy Reasoner (SSR) perform inference on global and local policies, respectively. PCR ensures that global regulations are not violated by the radio and provides SSR with opportunities for reconfiguration. SSR directly monitors and reconfigures the radio system based on the contextual information from the environment (via sensors) and according to the local policies. While the interface between SSR and PCR is defined by a Policy API [58], the interface between SSR and the SDR is not. Before a radio can process and respond to requests from PCR and SSR, it needs to be wrapped with a glue code, specific to the radio’s platform.

The XG program has been consecutively developed by three different companies: BBN Technologies, SRI International and currently by the Shared Spectrum Company. Each
of them defined their own language to express policies – XG Policy Language Framework (XGPLF)[59], Cognitive Radio (Policy) Language (CoRaL)[60] and XG Policy Language (XGPL)[61], respectively. Since only the latter is currently supported, we omit description of the previous languages. XGPL is a declarative language based on Web Ontology Language (OWL) Lite [46] and Semantic Web Rule Language (SWRL) [62]. Ontological concepts regarding SDR’s hardware and software components are defined in the XG ontology. While the operational parameters of the SDR can be stored by SSR in an arbitrary format, when information is passed from the SSR to the PCR, it needs to be expressed in terms of the XG ontology [61].

XG has been successfully implemented and deployed on several radio platforms. Most notably, its use was demonstrated in a scenario where XG nodes made up ad-hoc networks and avoided interference to existing non-XG radios by automatically adapting frequencies [16]. All radios in the scenario were identical and used a wrapper to control and monitor an IEEE 802.16 WiMAX Modem. It has been shown [57] that XG can be enabled on an IEEE 802.11 devices by extending them with additional software components and a sensor. Another demonstration of XG [58] was performed with six SSC DSA 20100 radios using the
IEEE 802.16 waveform and modulation. In this scenario, however, radios used heterogenous hardware and operating systems for the general purpose processing required to execute the XG reasoners. Successful implementations and field-testing of XG prompted several military radio platforms to integrate this DSA technology [41].

Although it is true that the XG platform is independent of the radio architecture, in practice, it means that each radio platform needs a custom layer of code that can process XG policies and translate them to radio and waveform-specific invocations. This results in XG-enabled architectures that utilize their own, non-standard APIs, e.g. [63]. From this perspective, the interface between XG and radio components is similar to the SDR-API approach, where each radio requires its own dedicated adapter. On the other hand, an open API for interfacing DSA functionality and radio platforms within the XG program has been proposed and implemented on several military radios [41]. This type of interface, however, suffers from all the drawbacks of the Standard API approach. Moreover, DSA is just one application of CR and a standard DSA API would not be sufficient to support all of the CR functionality.

Table 2.5 summarizes how the XG architecture fits in with the features of the CR architecture that we specified above. Because XG does not specify implementation details for the interfaces between its reasoners and the radio platform, it supports knowledge reusability and platform-independence, as long as they can process and adhere to XG policies. Because each radio platform needs to implement a specific glue code for interfacing XG, its support is limited to XG-enabled radios. Adapting to domain changes, such as appearance of new knobs and meters, would require recoding of the glue code layer.

### 2.3.6 Query-based interface

The last architecture included in the review is the Unified Link Layer API (ULLA) architecture [64], in which wireless communications links can be queried and configured in a
Table 2.5  XG architecture and its support for the desired features of the CR architecture

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reusable knowledge</td>
<td>Full</td>
<td>It has been demonstrated [16] that the same DSA rules can be executed on heterogeneous radio platforms.</td>
</tr>
<tr>
<td>SDR-Independent</td>
<td>Partial</td>
<td>XG can only be executed on radios that a custom glue code was written for.</td>
</tr>
<tr>
<td>Platform-Independent</td>
<td>Full</td>
<td>XG is not limited to any particular platform. It has been demonstrated [58] to work on heterogeneous hardware and operating systems.</td>
</tr>
<tr>
<td>Domain-adaptable</td>
<td>None</td>
<td>Similarly to the Standard API, introducing new functionality would require recoding the glue code.</td>
</tr>
</tbody>
</table>

device-independent fashion. This architecture is not really meant to support physical layer reconfiguration, rather it aims at providing upper layers with uniform access to contextual information gathered from the link layer. This information could be utilized in applications such as link-aware and adaptive multimedia streams or routing [33]. The architecture of ULLA comprises three major layers (Figure 2.9):

1. Link Provider Interface – interface to various wireless device drivers, which need to be wrapped with Link Layer Adapters. This interface is meant to be implemented by the wireless device vendors and it is expected that future devices might natively support it, making the adapters obsolete.

2. ULLA Core – middleware layer, which mediates between the users and the link providers. It is somewhat similar to the SCA’s Core Framework, which also provides hardware abstraction layer.

3. Link User (LU) Interface – device-independent interface to the link providers. Using this interface, users can issue query and commands to the link providers and register with them to receive notifications. These notifications provide a callback mechanism to notify registered users that link layer conditions have changed.
The most interesting part in this design is the use of a query language (UQL), subset of the Structured Query Language (SQL), in order to specify queries about the link-layer information, for instance [33]:

```
SELECT linkId, RSSI FROMullaLink WHERE state=CONNECTED
```

Unlike in the architectures described above, where software components invoke some API methods or exchange well-defined XML documents, ULLA provides them with a query language. Support for this query language requires implementation of moderately sophisticated adapters on each wireless device, so that queries can be translated into device-specific method invocations. An open-source reference implementation\(^6\) of ULLA is available, but it supports only two wireless adapters, is tightly integrated with the Linux operating system and sup-

\(^6\)Open-source reference implementation of ULLA is available at http://ulla.sourceforge.net/
ports only C/C++ applications.

According to its authors, ULLA could be used to realize the concept of CR without the need for SDR [64]. Although it certainly is true that ULLA could enable intelligent applications, it does not yet support adaptive reconfiguration of the physical layer, which is at the core of CR. If used with an inference engine, ULLA would require another software layer that would translate between ontological concepts and the UQL queries formed in the LU Interface. In addition, query capability offered by UQL is already provided by most inference engines, allowing higher-layer applications to query the knowledge base in a similar fashion. Unlike semantic query languages, such as SPARQL, UQL is not a standard query language and has a rather limited functionality.

If the CR architecture used ULLA, the LU interface would \textit{de facto} play the role of a standard API. Thus, the fulfillment of our requirements for a CR architecture by ULLA-based CR would be no different than that of the Standard-API approach.

### 2.3.7 Summary of the related work

None of the existing CR architectures fulfill all of the requirements that we stated above (see Table 2.6), nonetheless, some conclusions can be drawn:

- Designing a SDR-specific interface results in the least flexible architecture.
- Creating an open standard API results in a framework which supports the reuse of rules and ontologies across the platforms that implement it. This comes from the fact that the interface between the reasoner and the SDR platforms is \textit{fixed}. On the other hand, relying on standard APIs makes the architecture inflexible — a standard can only be updated as a result of an agreement of its stakeholders, but also once updated it must be followed by revisiting the implementation on both ends of the interface. This problem is exhibited by both the Standard API and the SCA-based architectures.
which are reusable, but not very flexible.

- There are different ways to support platform-independence:
  - use XML-based API, as in VTCE
  - incorporate platform-independence into the architecture with the use of platform-independent middleware, like CORBA
  - avoid dependence by defining more abstract architecture, as in XG

- In all of the reviewed architectures it is expected that the SDR either implements a radio-specific API, or that the API is implemented in a specific technology. None of the architectures guarantee complete SDR-independence.

- None of the reviewed architectures is fully flexible (adaptable). They all require recoding, recompilation and redeployment after a change has been made to a radio-specific API. Only OBR provides some flexibility due to the use of reflection, but at the same time it fails to support other features.

<table>
<thead>
<tr>
<th>Feature</th>
<th>SDR API</th>
<th>Standard API</th>
<th>SCA(^a)</th>
<th>OBR</th>
<th>XG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reusable knowledge</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>SDR-Independent</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Platform-Independent</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Domain-adaptable</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

\(^{a}\)Assuming that the SCA was augmented by a cognitive engine API

The review of related work reveals significant limitations and challenges of standardizing the interface between the reasoner and an SDR. Despite their obvious benefits, there are at least two drawbacks of using radio-specific standard APIs: 1) it takes a significant effort to change them, and 2) creating a new version that is not backwards-compatible results in losing the interface among the components that implement different versions of the standard.
The rest of this work describes our efforts towards solving this problem by proposing a new architecture that is capable of maintaining the interface with the SDRs that change their APIs without the need of recoding the reasoner API. This comes from the fact that the interface information is provided dynamically via the software model.
Chapter 3

Background Information

In this chapter we introduce concepts that are later used in the description of our CR architecture.

3.1 Knowledge Representation

The Cognitive Radio paradigm requires that the radio is equipped with awareness, or in other words it has the knowledge of its current internal state, of its own capabilities and of its environment. Moreover, the radio should also be able to share its knowledge with other cognitive nodes. When Mitola introduced the concept of the CR [8], he described a Radio Knowledge Representation Language (RKRL) that would serve this purpose. RKRL was never fully implemented, although it has been used in research, for instance, to show that it could enable a cognitive engine to deal with location awareness [65].

For the purpose of knowledge representation and reasoning, we chose the Web Ontology Language (OWL) [46] standard that has been widely adopted in the Semantic Web community. The semantics of OWL is based on Description Logics (DL), a family of Knowledge Representation (KR) formalisms equipped with a formal, model-based semantics [66]. The
architecture of a system based on DL (Figure 3.1) allows for setting up a knowledge base (KB) and for reasoning about its content. The KB consists of two components, TBox and ABox. In the context of OWL, the TBox introduces axioms defining the ontology — classes and properties; the ABox contains assertions about named individuals — instances of classes and properties defined in the TBox [66].

Figure 3.1: Architecture of a knowledge representation system based on Description Logics

In the context of the knowledge-based CR architecture, TBox consists of axioms about the radio domain, shared by all cognitive radios. ABox, on the other hand, represents facts pertaining to a particular radio. When radios exchange knowledge, they exchange ABox fragments, which can be understood via the TBox axioms.

The ontological knowledge itself is not sufficient for the cognitive radio applications, which also require rules to act as the "bridge" between conceptual knowledge and the problem-solving needs of a particular application [21]. Due to OWL limitations, not everything can be expressed in OWL [67] and the knowledge base must be augmented by RBox, i.e. a set of rules. Thus, one of the requirements in the choice of a reasoner is the support for reasoning over both OWL and rules.
3.2 Ontology matching

The focus of our work is on the interface between a knowledge-based inference engine (reasoner) and a software-defined radio. The reasoner can only process data that is expressed in ontological terms. Whenever two (or more) different ontologies with overlapping concepts are used together, there is a problem of how to correlate these concepts with each other. Methods from the field of ontology matching offer a range of solutions to this problem. Ontology matching is a well-advanced area of research and its techniques could potentially be applied to several aspects of the CR architecture (discussed below). Information in this section is largely based on [68] – a recent overview of the state of the art in the ontology matching research.

Ontology matching is defined as “the process of finding relationships or correspondences between entities of different ontologies”. The output of matching, called alignment, is a set of correspondences that express the relationship between two ontologies. Alignments include, but are not limited to, statements such as entity equivalence, sub-super relationship between entities, class intersection, or inverse relation.

Alignment can be used to generate tools used for further automated processing. For instance, Figure 3.2 shows a translator generated from the alignment in order to translate data instances from one ontology to another. Figure 3.3 shows a different example, in which alignment is used to generate a mediator that can translate queries expressed in one ontology to another, and translate answers in the opposite direction.

There are numerous applications of ontology matching. Below we list the most important ones along with ideas about how some of them could influence future CR research.

- Integration of information from heterogeneous data sources – could improve the process of collecting information about CR’s environment by incorporating data from sources that are not readily available in the ontological format, e.g. local database.
Figure 3.2: An example of using ontology matching to generate a data instance translator (based on [68])

- Matching overlapping concepts from imported ontologies
- Semantic peer-to-peer information sharing – could enhance peer-to-peer CR networks by allowing heterogeneous peers to exchange data
- Semantic web service composition
• Navigation and query answering on the web

When aligning ontologies, the matcher deals with different types of heterogeneity (listed in the order of complexity):

• **Syntactic** – ontologies are expressed in different syntax.

• **Terminological** – the same entities are differently named.

• **Conceptual** – the same concepts are defined using different axioms.

• **Semiotic** – concepts that have the same semantic meaning may be differently interpreted by humans depending on the context in which they appear.

Syntactic differences can be handled at a level where the knowledge about the meaning of the ontological concepts is not even required, and allow for fully automated matching. On the other hand, since it is very hard to detect and harness the semiotic heterogeneity, it lacks automated techniques at all. The research in ontology matching focuses on the terminological and conceptual types of heterogeneity, offering the following techniques (broken down based on granularity):

• **Element-level**: string-based, language-based, constraint-based, linguistic resources, alignment reuse, upper level and domain specific formal ontologies

• **Structure-level**: graph-based, taxonomy-based, repository of structures, model-based

The process of matching can be improved by combining different techniques together, resulting in chains of matchers. In addition, it can be enhanced by generic AI algorithms, such as machine learning or probabilistic methods. Ontology matching is usually performed once per application, resulting in an alignment document. It makes sense to repeat the process only if the ontologies change or a new matching technique is introduced.

Despite the use of sophisticated methods from AI, ontology matching can rarely be fully automated beyond relatively simple correspondences, covering syntactic and terminological heterogeneity. When complex relationships come into play (conceptual heterogeneity),
matching algorithms often have difficulties identifying any correspondences at all, or find ones that are irrelevant. For most methods, matching systems allow users to specify a threshold for confidence held in the correspondences. This allows the user to eliminate matches that are more likely to be invalid. Since 2004, Ontology Alignment Evaluation Initiative (OAEI) \(^1\) has been organizing a contest for comparing performance of different matching systems on well-defined test cases. Each year participating systems have been improving their performance, yet they never cover the entire alignment. That is not to say in less complex test cases some of them would not perform a full matching.

When the matching is incomplete, finishing the alignment manually is necessary, although even this task can be cumbersome. The manual alignment can be facilitated with the use of ontology alignment design patterns \(^69\), which stem from the observation that different sets of ontologies, even from completely different domains, exhibit similar types of complex relationships. Design patterns are particularly useful for finding solutions to complex relationships, e.g. a property in one ontology has the same intention as a relation in the second ontology, which requires transforming data values into specific class individuals.

### 3.2.1 Representing and processing alignments

*Expressive and Declarative Ontology Alignment Language* (EDOAL) \(^70\) is a language designed by the ontology matching community specifically to address the problem of expressing complex relationships between ontologies. It is the most recent version of a language that started off as a SEKT Mapping Language \(^71\) and has been gradually maturing over the years, with each version providing more support for complex alignments. The semantics of EDOAL is independent from any ontology language, which has two benefits – it can be used to match ontologies grounded in different syntax, and it allows for expressing design patterns at an abstract level. EDOAL alignments can be expressed using different syntaxes, the one

---

\(^1\)OAEI website: [http://oaei.ontologymatching.org/](http://oaei.ontologymatching.org/)
we will use is RDF/XML-based, since it has a close relationship with OWL.

EDOAL recognizes four types of entities: class, instance, relation and property. In the context of OWL this corresponds to class, individual, object property and datatype property, respectively. Each entity can be restricted and property values can be subject to transformations. Listing 3.1 shows a very simple example of RDF/XML-based EDOAL document. Each match is expressed as a Cell element, and each Cell contains references to entities from both ontologies, the relationship between them and a confidence value. By convention, EDOAL maps entities from the first ontology to the second one.

Listing 3.1 Basic structure of an EDOAL alignment document

```
<Alignment>
  <onto1>
    <Ontology rdf:about="http://example.org/o1">
      <location>file:examples/rdf/onto1.owl</location>
      <formalism>
        <Formalism align:name="OWL1.0" />
      </formalism>
    </Ontology>
  </onto1>
  <onto2>
    <Ontology rdf:about="http://example.org/o2">
      <location>file:examples/rdf/onto2.wsml</location>
      <formalism>
        <Formalism align:name="WSML" />
      </formalism>
    </Ontology>
  </onto2>
  <map>
    <Cell>
      <entity1 rdf:resource='http://example.org/o1#reviewedarticle'/>
      <entity2 rdf:resource='http://example.org/o2#journalarticle'/>
      <relation>Equivalence</relation>
      <measure>0.46</measure>
    </Cell>
  </map>
</Alignment>
```
The real strength of EDOAL, however, lies in its ability to refer to groups of entities using boolean operators (disjunction, conjunction, complement), property construction operators (inverse, composition, reflexive, transitive and symmetric closures) and constraints (domain, range, cardinality and value restrictions) [72]. For instance, hasUncle relation in one ontology can be aligned with a composition of hasParent and hasBrother in the second ontology, as shown in Listing 3.2.

**Listing 3.2** Representing more complex relationships in EDOAL

```xml
<Cell>
  <entity1>
    <edoal:Relation rdf:about='http://ex.org/o1#hasUncle'/>
  </entity1>

  <entity2>
    <edoal:Relation>
      <edoal:compose rdf:parseType='Collection'>
        <edoal:Relation rdf:about='http://ex.org/o2#hasParent'/>
        <edoal:Relation rdf:about='http://ex.org/o2#hasBrother'/>
      </edoal:compose>
    </edoal:Relation>
  </entity2>

  <relation>Subsumes</relation>
  <measure>1</measure>
</Cell>
```

Regardless of how the alignment was created – automatically by matcher, manually or partially by both methods – it can be used to generate translators and mediators. In the context of OWL, an instance translator can be implemented as an XSLT script and mediator can be realized as a set of OWL axioms that represent the correspondences in the alignment. EDOAL would certainly be of lower value if it was not accompanied by Alignment API [73], a comprehensive Java API for manipulating alignments. Alignment API was created around the time the first predecessor of EDOAL was designed. It aims to cover functionality related to the ontology matching process as a whole by providing abstractions for matchers, evaluators, renderers and parsers. Using the API, alignment documents can be parsed and
then rendered to generate XSLT scripts, OWL axioms, etc. In its most recent version [72],
support for processing EDOAL was added, although included renderers are fairly limited at
this moment, e.g. the OWL axioms renderer does not take advantage of any of the features
added in OWL 2 [74]. This limitation can be addressed by implementing custom renderers.

Although complete automatization of the ontology matching process still has a long way
to go – and perhaps can never be fully realized – EDOAL and Alignment API form a solid
platform for realizing matching use cases.

3.3 Knowledge exchange

As intelligent agents, Cognitive Radios should have the ability to ”talk” to each other in
order to take and coordinate actions. Some CR use cases require that radios have the ability
to exchange knowledge with other CR nodes, e.g. the Network Extension [45] use case, in
which radios build ad-hoc networks in emergency situations.

When Mitola first introduced the concept of CR, he described a RKRL language [8]
intended to be used for representing radio’s knowledge, but also for exchanging it with
other CRs. In order to support the knowledge exchange, RKRL incorporates Knowledge
Query Markup Language (KQML)[75], an Agent Communication Language (ACL) that was
designed to support interactions between intelligent software agents. KQML relies on the
speech act theory [76] and separates the contents of each message from its speech act, also
known as performative. The performatives can either be assertive – used for asserting facts,
or directive, e.g. query or command. The content of a KQML message can be arbitrary
and its syntax is not part of the language, although KQML allows specifying annotations
about the content, e.g. its language and assumed ontology, which allows other agents to
understand it.

Another ACL that relies on the speech act theory is FIPA-ACL – part of the Foundation
for Intelligent Physical Agents (FIPA) architecture. FIPA is a standard specification of an abstract architecture for intelligent multi-agent systems, developed and maintained by an international not-for-profit organization, also named FIPA. The specification of FIPA is abstract – it permits multiple concrete realizations, although it supports interoperability and reusability. The interoperability between different implementations is achieved by including agent management and message transport specifications in the standard.

FIPA-ACL and KQML share a lot of similarities, including syntax, but differ in the set of supported speech-acts and significantly in the language semantics [78]. FIPA-ACL has formal semantics, which eliminates ambiguity and supports interoperability. KQML initially lacked formal semantics, and it was not provided until a few years later. Unlike KQML, which currently lacks organized effort towards further development and maintenance, FIPA is backed by standards and an organization comprising a number of private and academic institutions. More importantly, different implementations of KQML were reported to have difficulties with interoperability [78]. Finally, KQML lacks tool support with a large user base. Although initially there have been several implementations of FIPA specification, currently one is actively developed i.e. Java Agent DEvelopment Framework (JADE) [79]. JADE is an open-source FIPA-compliant agent platform developed in Java and recommended by the author of KQML himself 2.

The FIPA standard includes a library of 22 communicative acts, out of which only one needs to be implemented by an agent to be FIPA-compliant, namely not-understood. The standard also includes a library of interaction protocols, which are common patterns of message exchanges, such as Query, Request or Dutch Auction. All protocols are defined in terms of speech acts from the library with the Agent UML (AUML) [80] sequence diagrams. AUML is an extension of the UML, which for the most part became obsolete with the release

of UML 2.1 and SysML. Figure 3.4 shows a sequence diagram representing the FIPA Query interaction protocol. The diamonds in the diagram represent alternative paths, along which the interaction may occur – this element is not part of the standard UML.

![Sequence Diagram of FIPA Query Interaction Protocol](image)

**Figure 3.4:** FIPA Query Interaction Protocol represented using an AUML sequence diagram (Source: [77])

Authors of FIPA chose sequence diagrams for representing protocols to emphasize the agent-centric view, as opposed to state machines, which are state-centric [80]. Each protocol is accompanied by an exception, which can occur at any time and terminate the entire protocol. This exception occurs when the receiver returns a **not-understood** message. The interaction protocols are entirely domain-independent – they only describe the order in which the speech acts can occur, but they do not say anything about the content of the exchanged messages. Obviously, custom protocols can be designed, but this goes beyond the standard and carries the risk of losing interoperability with other agents.
3.3.1 Agent Communication Languages and CR networks

*Cognitive Network* is a term used to describe a network that has a cognitive process following the OODA cycle with the network-level objectives [81]. A cognitive network built on top of CR nodes is termed *Cognitive Radio Network* (CRN). Although somewhat closely related, CRN differs from CR in the scope of the goals that are under control – CR goals are local and user-oriented, CRN goals are global, network-oriented and focus on the end-to-end network performance [82]. Note that CRs may form networks that are not cognitive if there are no network-level objectives. The topic of CRN is subject to very active research [83, 84], although there is little work that borrows directly from the multi-agent system architectures.

Both FIPA and KQML assume that all agents are connected and can communicate with each other in a reliable fashion, preserving the order of messages – this is a serious challenge in the wireless environment, which is inherently unreliable. While for point-to-point connections these assumptions may be correct, they are limiting in dynamic and complex CRNs [85] with rapidly changing channels and mobility, which may result in a frequently changing context.

Mähönen [86] indicated that Mitola's RKRL (and by extension KQML) is not sufficient enough to build CRNs and suggested extending it to a Network Knowledge Representation Language (NKRL), which would allow for expressing high-level goals of the operators. On top of this idea the concept of a Cognitive Resource Management (CRM) architecture [87, 88] was developed with the intention to build a foundation for implementing CRNs. CRM is a component with a cross-layer access to radio via ULLA interface (described in Section 2.3.6), and a toolbox comprising various AI methods that can aid the decision-making process. CRM architecture makes no use of the ACLs.

Friend et al. [89] argue that cognitive network, and by extension CRN, is more suitable for applying solutions from the area of cooperative distributed problem solving (CDPS), rather than from multi-agent systems where agents tend to be autonomous. CDPS is better aligned with the requirement of the existence of end-to-end goals, *shared* by all agents. Nevertheless,
CDPS methods, such as distributed constraint reasoning, distributed Bayesian networks or parallel optimizations (genetic algorithms, tabu search, etc.) can be implemented within a multi-agent system framework. Regardless of how they are implemented, however, CDPS methods need to be adjusted to take into account the lack of data reliability in the wireless environment.

In this work we do not focus on CRNs, rather we focus on networks of CRs, where each radio aims to achieve local goals. These local goals may require several CRs to communicate with each other, but they are never end-to-end goals of the entire network.

3.4 CORBA Middleware

To meet the requirement of platform-independence, our architecture makes use of CORBA. We provide a very brief description of this middleware, mostly to introduce terms that we will be referring to later, in the description of the architecture.

Each component within the CORBA middleware must publish its API using the Interface Definition Language (IDL), which allows for describing software components' interfaces in a language-neutral fashion. Communication in CORBA follows the client-server paradigm (Figure 3.5) – components that provide services act as servers, and service requestors as clients.

![Figure 3.5: Client-Server communication using CORBA middleware](image)

60
To enable communication, client and server must use the same IDL. Before IDL can be implemented by a component, it needs to be compiled to one of the languages for which a mapping has been defined by the OMG. With the use of tools, the IDL is compiled into stubs on the client side, and into skeletons on the server side. Client invokes methods on the local stubs, which serialize the request into a message that is passed to Object Request Broker (ORB). ORB, which effectively constitutes an implementation of the CORBA specification, transmits the message to its destination (local or remote). On the server side, skeleton deserializes the message, locally invokes the method and returns the result to ORB. ORB transmits the result over to the client, and the invocation ends.

The platform-independence of CORBA stems from the variety of supported languages and the number of platforms on which ORB implementation is available. From the developer's perspective, the greatest benefit of using CORBA comes from the fact that the client and server can be implemented independently, using different languages and running on different hardware and software platforms – as long as they use the same IDL. In this respect, they are very similar to Web Services – we elaborate on the choice of CORBA when discussing our architecture.
Chapter 4

Cognitive Radio Framework (CRF)

In this thesis a novel Cognitive Radio Framework (CRF) architecture is proposed. It is a platform-independent extension of the OBR architecture. CRF aims to meet all of the requirements stated in Section 2.2. Instead of relying on radio-domain specific APIs to interface the reasoner, CRF provides a thin, generic API for connecting with the inference engine that uses the software model of the radio platform to identify access paths to particular radio parameters (knobs and meters). This architecture is capable of maintaining the interface with the SDRs that change their APIs without the need of recoding the reasoner API due to the fact that the interface information is provided dynamically via the software model. Unlike OBR, the CRF architecture does not store SDR-specific information in the rules, which enables reuse of the rules across different radio systems.

4.1 Architecture

The CRF architecture (Figure 4.1), although initially designed as an extension of the OBR, introduces numerous improvements and differs significantly from its predecessor:

• It incorporates CORBA middleware to guarantee platform independence when inter-
facing SDR

- It implements some of the FIPA standard interaction protocols that together form a state machine, which reacts to messages coming from other radios
- It includes support for a reliable data transfer, which is particularly important for the exchange of control messages
- It communicates with the SDR using a domain-independent LiveKB component that offers a generic API to access radio’s functionality

**Figure 4.1**: The Cognitive Radio Framework (CRF) architecture

The CRF architecture can be viewed as a composition of three parts: Message Layer, Cognitive Layer and LiveKB. We begin the description of CRF by describing each layer in the following sections.
4.1.1 Message Layer

Numerous CR use cases [23, 24, 45] involve communication between multiple nodes. For instance, two cognitive radios negotiate the structure and the length of the equalizer’s training sequence in order to reduce the packet overhead, thus improve the bandwidth [23]. Implementing such use cases requires that the cognitive engine architecture facilitates the exchange of control messages without interfering with the user application’s data/voice stream. Message Layer serves this purpose in the CRF architecture.

In the OBR architecture, from which CRF originated, incoming and outgoing packets are recognized as one of the five types: Data, Confirm, Query, Answer and Command (cf. Figure 4.2). The Data messages refer to the user application stream, and the remaining four pertain to the inter-radio exchange of control information. In CRF this breakdown was simplified and only two types of messages are distinguished at the highest level: Data and Control. Since CRF incorporates elements of the FIPA architecture (cf. Section 3.3), Control messages correspond to the library of FIPA communicative acts, but these are not distinguished until they reach Cognitive Layer. Thus, Message Layer identifies only Data and Control types.

![Figure 4.2](image)

**Figure 4.2:** Types of messages distinguished in the OBR architecture
In contrast with most protocols where packet types are recognized by their frame, the two types of messages in CRF are encoded entirely within the packet’s payload, as it was done in OBR. This allows for a great flexibility in terms of the size and content of control messages, but also supports compatibility with existing packet frames.

Message Layer comprises two components: Data In/Out (DIO) and Data Front-End (DFE). DIO is the central component of Message Layer, it works as a packet switch and directs data messages to and from DFE, and control messages to and from the Cognitive Layer. Messages are passed asynchronously and buffered to compensate for the difference between the speed of message production and consumption. DIO recognizes the type of incoming messages based on their content, although they do not need to be fully parsed for this purpose. For instance, message type can be indicated by a bit sequence prefix and since only two types of messages are distinguished in CRF, a single bit would suffice to represent a type. The DFE component is available via CORBA and allows upper layer applications to connect to CRF in a platform-independent fashion. After registering a callback method with
DFE, applications can send and receive data without interfering the rest of the architecture.

In order for the cognitive radios to communicate with each other, control messages must be delivered in a *reliable* fashion. Since data transmission using the SDR is usually implemented with the support of well-established network protocols, DIO can take advantage of their support for reliability. The lowest, physical layer of the network stack, is unreliable and does not involve any data error checking, however, the next layer in the stack, Data Link (DL) layer, does implement one of the Reliable Data Transfer (RDT) protocols. In summary, there are two options to guarantee a reliable exchange of control messages:

- Interface DIO with the SDR’s physical layer and implement an RDT protocol within the DIO component.
- Interface DIO with SDR at a layer that already provides RDT.

Under the OODA loop framework (cf. Section 1.2.1), Message Layer participates in the Observe and Act processes by the virtue of control messages exchange. Incoming messages may contain knowledge sent from other radios, thus aiding the process of observation. On the other hand, it is an actuator when transmitting outgoing messages to other radios.

### 4.1.2 Cognitive Layer

The Cognitive Layer, which is the core part of CRF, follows the knowledge-based approach to implementing the cognition cycle and is primarily responsible for the orienting and the deciding processes of the OODA loop. It consists of two components – a *Monitor Service* (MS) and a *Reasoning Component* (reasoner). Since the reasoner requires axioms and facts to be already present in the Knowledge Base (KB) before it can operate, KB must be first populated and this task is handled by MS. MS, which effectively acts as the reasoner’s controller, is driven by a finite state machine (FSM) that implements standard FIPA Interaction Protocols.
FIPA FSM may transition to a new state in reaction to the following types of event – an incoming ACL message sent from a different radio, an outgoing ACL message produced by the reasoner, or a locally defined trigger that starts the inference, e.g. timeout. Depending on the current state, MS may invoke the reasoner or send an ACL response message without calling the reasoner, e.g. send failure when the incoming message cannot be processed in the current state.

Before MS can invoke the reasoner, it needs to fetch ABox facts representing the SDR’s current state. For this purpose, MS uses the LiveKB component, which collects values of all meters from the radio and returns them in the ontological format. Along with the TBox, facts and rules sent from other radios (if any), ABox facts are asserted into the reasoner’s KB and the reasoning is executed. As a result of inference, some procedural attachments might be invoked from within the rules. These attachments may set knobs on the radio using LiveKB, or generate ACL messages intended to be sent by MS to other radios. Depending on the outcome of inference, FIPA FSM may transition to a new state and ACL messages might be forwarded to the DIO component.

In summary, it is the combination of FIPA FSM and the rules (with procedural attachments) that drive the cognitive behavior of CRF. We will elaborate on this process further in the remaining sections of the chapter.

4.1.3 LiveKB

The LiveKB component [90, 91] represents arguably the most significant contribution of the CRF architecture, for the reason that it provides an entirely generic interface for accessing SDR’s parameters. LiveKB allows MS and the reasoner to read SDR’s meters and set its knobs without knowing the SDR’s API at design time. Before it can be used, LiveKB has to be dynamically configured to work with a particular SDR. For this purpose, it requires that the following are provided:
1. The CR ontology
2. IDL model of the radio, which by the means of annotations is aligned with the CR ontology
3. Name of the SDR’s root object registered with the CORBA Naming Service

Once configured, LiveKB translates requests from the cognitive layer into SDR-specific method invocations, which are then executed using reflection. We explain this process in the next section.

In order to support platform independence, LiveKB is implemented as a CORBA component. Since a new LiveKB object needs to be configured at runtime for each SDR, and because new CORBA objects can only be created on the server side, LiveKB Object Factory must also be present. As a consequence, at bootstrap, before the cognitive layer can start interacting with the SDR via LiveKB, the object factory must be first invoked to create an instance of LiveKB.

4.2 LiveKB Design

The greatest benefit of using LiveKB to access knobs and meters is the fact that the radio does not have to implement any specific API. It must, however, respect some architectural constraints. In this section we explain the motivation behind LiveKB and the details of its design.

4.2.1 Motivation

The original idea behind designing the LiveKB component was to allow the reasoner to express its requests to read and write radio’s parameters exclusively in ontological terms. These terms are shared by all radios through a standard ontology and do not pertain to any specific API. Instead of having a number of methods that correspond to different parameters,
we would like to be able to use only two of them:

\[
\begin{align*}
&\text{get(propertyName)} \\
&\text{set(propertyName, newValue)}
\end{align*}
\]

For instance, if \texttt{hasTxAmplitude} and \texttt{hasCarrierFrequency} are datatype properties defined in the CR ontology, in order to get or set a value of these parameters in a radio, instead of invoking radio API-specific methods like \texttt{get\_txAmplitude()} and \texttt{set\_CarrierFrequency(2400)} we could invoke the following: \texttt{get(“hasTxAmplitude”)} and \texttt{set(“hasCarrierFrequency”, 2400)}, respectively.

In a sense, the CR ontology becomes the standard in this scenario. However, because it represents domain knowledge, rather than a programming interface, it is far less likely to change in the future than API. What is more important is the fact that the generic API allows for writing rules (policies) that are reusable, because the procedural attachments corresponding to get and set methods, used within the rules, are independent of the SDR software structure. Moreover, changes made to the ontology would only require changes in the rules, leaving the implementation of API intact.

### 4.2.2 Enabling generic API

The generic API requests invoked from within the rules can be processed in two ways:

1. Directly invoked on the radio.
2. First \textit{translated} to radio-specific methods and then invoked on the radio.

The first approach would impose a substantial requirement on each radio, because the ontology would have to be known at design time and become part of the implementation. This defeats the purpose of a generic API, because radios would need to recode their interface each time ontology changes.
The second approach allows the radio to provide its own API, yet keep the rules reusable. The crucial part of this design is translation, or mapping, from generic to radio-specific methods. A straightforward solution, similar to the ones used in most of the reviewed architectures, would be to define a standard API for all radios, then implement a layer of code that translates between get/set to the standard. As we indicated in the review, there are multiple problems with a standard API, such as a lack of consensus, a slow rate of changes, and problems with backwards compatibility. This is where the LiveKB component comes into play – instead of translating generic requests using a radio API, it does so dynamically at run-time using reflection, regardless of what API the radio provides.

The difference between invoking radio methods using its API and invoking them via LiveKB is shown graphically in Figure 4.4. Note that when using LiveKB, there is no need to know anything about the radio’s API on the reasoner side, and at the same time the radio can implement its own API. The benefits of this design are twofold – the reasoner can theoretically access any radio, and the radios do not have to implement any standard API.

![Figure 4.4: Accessing SDR using the LiveKB component](attachment:image.png)
4.2.3 Ideal design of LiveKB

The primary goal of LiveKB is to dynamically translate requests that come from the reasoner to method invocations that are specific for the radio. The fundamental concept for implementing this goal is ontology matching. Figure 4.5 shows the general idea. SDR, accessible via CORBA, provides its API as an IDL, which is automatically translated into its equivalent IDL ontology expressed in OWL. The generated IDL ontology is matched with the CR ontology by Matcher, whose output is passed to Generator to generate a mediator. At runtime, the mediator translates the requests coming from the reasoner to requests expressed in the IDL ontology. The axioms in the IDL ontology provide sufficient information to reflectively invoke the requested method on the SDR. Once an invocation is complete, the mediator translates the result of the invocation into terms of the CR ontology and sends it back to the reasoner.

![Figure 4.5: Ideal design of LiveKB](image)

The first three steps – generation of the IDL ontology, matching it with the CR ontology...
and generation of the mediator from the alignment, need to be done only once, at startup. After that, LiveKB can translate reasoner’s requests by utilizing the artifacts produced earlier, and mostly performs reflective invocations.

The realization of the conceptual design of LiveKB (Figure 4.5) is very challenging since it depends on automatization of three somewhat complex tasks:

1. Translation of an IDL model into an OWL ontology
2. Ontology matching between IDL and CR ontologies
3. Generation of a mediator from the alignment

The first task can be automated, as long as the IDL respects some constraints (we will discuss this later). However, for the reasons explained in Section 3.2, ontology matching is not ready to be fully automated for complex correspondences. As a consequence, manual or semi-manual matching is necessary to be done for each radio. Furthermore, since this field is relatively young, the tool generators for EDOAL alignments are still immature and thus this process also cannot be fully automated. Consequently, at this point we cannot fully realize the LiveKB design as depicted in Figure 4.5. Nonetheless, given the progress done each year in the ontology matching field, it is anticipated that an automatic ontology matching will be realizable to a higher degree in the near future.

### 4.2.4 Feasible design of LiveKB

Since the matching cannot be fully automated yet, it needs to be done manually, or somehow assisted to produce a full alignment. In order to facilitate this process, we altered the LiveKB design and made it feasible for implementation. It allows for dynamic translation between generic and radio-specific API, but requires more input from the SDR vendor. The revised, more feasible LiveKB design is shown in Figure 4.6.

In this design, not only SDR must be available via CORBA and provide its IDL, but
the IDL must also be *annotated* to aid the matching process. The annotated IDL provides enough information for the *Assisted Matcher* to create a full alignment between the given CR ontology and the IDL ontology generated within LiveKB from the SDR IDL model. The matcher also generates an *Invoker*, which can execute SDR methods represented as properties in the IDL ontology with the use of the reflection mechanism.

Since the generation of tools solely based on alignment is still limited, we use alignment only to generate *bridge axioms* [68], which *merge* the two ontologies together. Bridge axioms can be easily generated, because they correspond to the alignment almost in a one-to-one fashion. Using rules and bridge axioms, requests formulated by the reasoner in terms of the CR ontology can be automatically translated into terms of the IDL ontology. A request expressed in the IDL ontology provides sufficient information for the Invoker to locate an
object in the SDR runtime and execute an appropriate method. Invoker can also read all
the parameters and represent a radio’s current state as a collection of CR ontology ABox
assertions.

If we treat LiveKB as a black-box, it needs to be provided with (1) an annotated IDL
model of the SDR and (2) the CR ontology. At bootstrap, it produces (1) an IDL ontology
and (2) bridge axioms that need to be loaded into the reasoner’s KB. At runtime, it can
produce a CR ontology ABox, which represents the SDR’s current state and can respond to
get/set requests invoked from within the reasoner’s rules.

A simple rule that invokes a setter to change a value of a parameter, given that some
condition is met, is shown as Algorithm 4.1 using pseudocode. Note that the rule writer is
not required to know the name of the setter property in the IDL ontology – it is found by
the reasoner using the bridge axioms.

**Algorithm 4.1 A simple rule that invokes a setter**

1: if $k \geq k_{max}$ then
2: $setter \leftarrow$ find a setter property in $O_{IDL}$ that is equivalent to $k$ in $O_{CR}$
3: `INVOKER(setter, newValue)`
4: end if

An alternative design could involve running a separate reasoner inside the LiveKB, which
would be used only for finding the equivalent properties. In this design the KB of the
reasoning component outside of LiveKB would not have to be augmented with the IDL
ontology and the bridge axioms. Although the rules would be a little more succinct (the
step on line 2 would be skipped), the CR as a whole would have to be running two reasoners
for each request, which is inefficient and unnecessary.

The feasible design of LiveKB involves generation of three artifacts: an IDL ontology,
alignment axioms and an an Invoker tool. In the following sections we describe how each of
these artifacts is generated in more detail.
4.2.5 Generating an IDL Ontology

The target language for representing knowledge in CRF is OWL (cf. Section 3.1). The purpose of creating the OWL IDL ontology is to represent IDL setter and getter methods in the ontological format that can be used during matching and when finding the methods from within the rules.

OMG, the standards body behind IDL, provides an Ontology Definition Metamodel (ODM) [92], which allows for modeling ontologies with UML. There are several benefits of using ODM: (1) it allows for representing ontological knowledge in a format that is independent of knowledge representation languages, such as OWL or WSML, (2) it provides mappings to OWL, and (3) knowledge engineers can make use of a variety of existing UML tools to design ontologies. Early on in our research we considered using UML as the representation of the SDR API, since UML is arguably the most common language to model software and because of the UML to OWL mapping support already provided by ODM. Unfortunately, the mapping turned out to be inappropriate for our needs – it is suitable only for UML diagrams representing ontologies, developed with the use of ODM UML profiles. As such, it is not possible to translate arbitrary UML diagrams provided by SDR vendors using ODM. Translating arbitrary UML to OWL cannot be currently supported due to the fact that UML lacks semantics. For this reason, we decided to use IDL for representing SDR API.

Not all elements of the IDL language are supported or needed for the translation. From our perspective, IDL is primarily a collection of interfaces, which in turn comprise attributes and operations. First we discuss which IDL datatypes are supported in our translation, then we show how the three primary constructs of IDL are translated into OWL.
Supported Datatypes

The IDL language offers a range of typical basic data types, but it also provides a wide variety of mechanisms to create user-defined data types. All IDL types can be divided into the following groups [93]: basic types, constructed types, template types, arrays, native types, interfaces, and value types. Since our translation is focused on methods that have access to parameters with concrete values, we narrow the translation support to basic types, string types (subclass of template types) and interfaces, which are the fundamental types that allow for describing services offered by CORBA objects.

In this work, we refer to the basic and string types as primitive types, and to interfaces as object types. While object types in IDL correspond to OWL classes, primitive types correspond to simple XML Schema types, which comprise OWL’s built-in data types. Table 4.1 shows how primitive IDL types correspond to XML Schema types.

**Table 4.1** Correspondence between IDL primitive types and XML Schema types

<table>
<thead>
<tr>
<th>IDL primitive type</th>
<th>XML Schema type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>xsd:boolean</td>
</tr>
<tr>
<td>char, wchar, string, wstring</td>
<td>xsd:string</td>
</tr>
<tr>
<td>double</td>
<td>xsd:double</td>
</tr>
<tr>
<td>float</td>
<td>xsd:float</td>
</tr>
<tr>
<td>octet</td>
<td>xsd:unsignedByte</td>
</tr>
<tr>
<td>long</td>
<td>xsd:int</td>
</tr>
<tr>
<td>long long</td>
<td>xsd:long</td>
</tr>
<tr>
<td>short</td>
<td>xsd:short</td>
</tr>
<tr>
<td>unsigned short</td>
<td>xsd:unsignedShort</td>
</tr>
<tr>
<td>unsigned long</td>
<td>xsd:unsignedInt</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>xsd:unsignedLong</td>
</tr>
</tbody>
</table>

Interface Declarations

An interface is the fundamental building block of IDL; it allows for defining services that are offered by the CORBA server. Every IDL model must have at least one interface declared. In our translation, an interface declaration is translated to an OWL class with the same
name. IDL also supports inheritance of interfaces and this is directly translated into the subclass relationship between the OWL classes that represent both interfaces. For instance, the following declaration

```idl
interface name : super
```

is translated into two axioms in OWL:

```owl
name rdf:type owl:class
name rdfs:subClassOf super
```

Attributes

We consider only those attributes that are of a primitive type, or of an array of primitive types. When IDL is mapped to a programming language, every attribute is associated with a getter method. If there is no optional `readonly` keyword preceding the attribute’s declaration, a setter method is also created for it. IDL attributes offer the most straightforward access to knobs and meters and are represented as a datatype property in OWL. The domain of the property corresponds to the interface within which the attribute is declared. The range of the property is one of the XML Schema types corresponding to the `type` of attribute (cf. Table 4.1).

If the name of an OWL property was exactly the same as that of the attribute for which it was generated, we would need to restrict each attribute name to be unique in the global scope of the IDL document. This stems from the fact that entity names in OWL are unique within the same namespace and if two different interfaces had attribute with the same name, the generated property would be ambiguous. For that reason, when generating any property in OWL, we name it with a concatenation of the interface name, the attribute/operation
name, and a @ sign between them. The choice of this symbol is not accidental – it is an illegal character for identifiers in IDL, but a legal character for entity names in OWL. This is important, because the Invoker will be able to separate the interface name from the attribute/operation name in a deterministic way.

For instance, assuming that the following attribute declaration occurred within the interface \textit{I}:

\begin{quote}
\textbf{[readonly]} attribute \textit{type name}
\end{quote}

it would be translated into the following OWL axioms:

\begin{align*}
I@\text{name} & \text{ rdf:type } \text{owl:DatatypeProperty} \\
I@\text{name} & \text{ rdf:type } \text{owl:FunctionalProperty} \\
I@\text{name} & \text{ rdfs:domain } I \\
I@\text{name} & \text{ rdfs:range } \text{type}
\end{align*}

If the \textit{type} was an array of primitive types, the same axioms would be generated, except for the second one, which indicates that the property can have only one value for each individual.

Note that regardless whether the attribute is \textit{readonly} or not, the same axioms are generated. We need to somehow indicate in the ontology whether the property relates to a getter and also to a setter, or not. For this purpose, we defined two datatype properties that are part of \textit{every} IDL ontology:

\begin{align*}
idl : \text{getterMethod} & \text{ rdf:type } \text{owl:DatatypeProperty} \\
idl : \text{setterMethod} & \text{ rdf:type } \text{owl:DatatypeProperty}
\end{align*}
In order to indicate whether a property represents a getter or a setter, a sub-property axiom must be added to the ontology. Assuming that the example attribute above is not readonly, the translator would add the following axioms:

\[ I \text{name} \ rdfs:subPropertyOf \ idl:getterMethod \]

\[ I \text{name} \ rdfs:subPropertyOf \ idl:setterMethod \]

The knowledge about this sub-property relationship is important when writing rules. Recall that in the basic rule expressed above in the pseudocode (Algorithm 4.1), the reasoner must find a setter property. This is accomplished by finding a property that is both equivalent to the CR property and is a subproperty of the idl:setterMethod property.

Attributes that are of an object type (instances of other interfaces) are not included in the translation to OWL, because they do not directly represent knobs or meters. Nevertheless, they could be easily represented as object properties with very similar axioms:

\[ I \text{name} \ rdf:type \ owl:ObjectProperty \]

\[ I \text{name} \ rdf:type \ owl:FunctionalProperty \]

\[ I \text{name} \ rdfs:domain I \]

\[ I \text{name} \ rdfs:range \ type \]

Similarly to the primitive types, we could define two object properties that represent a getter and a setter, and add sub-property axioms to denote whether a particular attribute is readonly or not.

The reason object attributes are not included in the translation is that rules will never directly access objects, they will only access values or change values, and these must be represented by primitive type attributes or operations with a primitive return type. We will
make use of the object type entities when generating the Invoker, since it must know how to \textit{reach} objects that contain primitive values (discussed below).

\textbf{Operations}

The automatic translation currently supports only those operations that either have a primitive return type and no parameters, or have a \texttt{void} return type and a single parameter of a primitive type. Similarly to attributes, they relate to single properties in the OWL ontology.

An operation without any parameter can only relate to a getter method, thus for an operation of the form:

\begin{verbatim}
    type name()
\end{verbatim}

the translation produces the same axioms as for the \texttt{readonly} attribute with a \texttt{type} and a \texttt{name}, as long as the \texttt{type} is not \texttt{void}.

Supported IDL operations with a single parameter have the form of:

\begin{verbatim}
    void name([in|out|inout] type parameterName)
\end{verbatim}

Parameters of IDL operations can be prefixed with one of the following three keywords, each of them indicating something else for the translator:

- \texttt{in} – client provides the value of the parameter – it is a setter method
- \texttt{out} – server provides the value of the parameter – it is a getter method
- \texttt{inout} – both client and server provide the value of the parameter – not supported

The \texttt{inout} direction is not supported by the translation, because it is not clear what type of a method it is. Every time such a method is invoked, client must provide a value, which could indicate a setter. However, the invocation always returns a result, which would indicate a
getter.

IDL operations with a single parameter are translated into OWL using the same axioms as for the attribute with a name and a type. Depending on the direction in which the parameter is passed, a sub-property axiom indicating a setter (in direction) or a getter (out direction) is also asserted.

Table 4.2 gives the summary for the IDL constructs that are supported by the automated translation to OWL.

Table 4.2  IDL constructs supported by automatic translation to OWL

<table>
<thead>
<tr>
<th>IDL construct</th>
<th>OWL Axioms</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface I</td>
<td>(I rdf:type owl:class)</td>
</tr>
<tr>
<td>interface I : S</td>
<td>(I rdf:type owl:class)</td>
</tr>
<tr>
<td></td>
<td>(I rdfs:subClassOf S)</td>
</tr>
<tr>
<td>context: interface I, type is primitive</td>
<td>(name rdf:type owl:DatatypeProperty)</td>
</tr>
<tr>
<td></td>
<td>(I@name rdf:type owl:FunctionalProperty)*</td>
</tr>
<tr>
<td></td>
<td>(I@name rdfs:domain I)</td>
</tr>
<tr>
<td></td>
<td>(I@name rdfs:range type)</td>
</tr>
<tr>
<td>attribute type name</td>
<td>(I@name rdfs:subPropertyOf idl:getterMethod)</td>
</tr>
<tr>
<td></td>
<td>(I@name rdfs:subPropertyOf idl:setterMethod)</td>
</tr>
<tr>
<td>context: interface I, type is primitive</td>
<td>(I@name rdf:type owl:DatatypeProperty)</td>
</tr>
<tr>
<td></td>
<td>(I@name rdf:type owl:FunctionalProperty)*</td>
</tr>
<tr>
<td></td>
<td>(I@name rdfs:domain I)</td>
</tr>
<tr>
<td></td>
<td>(I@name rdfs:range type)</td>
</tr>
<tr>
<td>Any of the following:</td>
<td>(I@name rdfs:subPropertyOf idl:getterMethod)</td>
</tr>
<tr>
<td>- readonly attribute type name</td>
<td>(I@name rdf:type owl:DatatypeProperty)</td>
</tr>
<tr>
<td>- void name(out type value)</td>
<td>(I@name rdf:type owl:FunctionalProperty)*</td>
</tr>
<tr>
<td>- type name()</td>
<td>(I@name rdfs:domain I)</td>
</tr>
<tr>
<td></td>
<td>(I@name rdfs:range type)</td>
</tr>
<tr>
<td>context: interface I, type is primitive</td>
<td>(I@name rdfs:subPropertyOf idl:getterMethod)</td>
</tr>
<tr>
<td>void name(in type value)</td>
<td>(I@name rdf:type owl:DatatypeProperty)</td>
</tr>
<tr>
<td></td>
<td>(I@name rdf:type owl:FunctionalProperty)*</td>
</tr>
<tr>
<td></td>
<td>(I@name rdfs:domain I)</td>
</tr>
<tr>
<td></td>
<td>(I@name rdfs:range type)</td>
</tr>
<tr>
<td></td>
<td>(I@name rdfs:subPropertyOf idl:setterMethod)</td>
</tr>
</tbody>
</table>

*a*This axiom is not asserted when the type is an array of primitive types.

Example

Listing 4.1 shows a sample IDL interface and Figure 4.7 shows the OWL ontology that was generated from that IDL. Note that all properties in OWL represent methods that can
potentially be used to invoke getters or setters. If the IDL model included attributes or operations that are not supported by the translation, they would be simply ignored – as is the case with the object attributes transmitter and signalDetector.

Listing 4.1   IDL interface of a trivial radio.

```idl
module api {
    interface SignalDetector {
        attribute float sampleRate;
    };

    interface Transmitter {
        float getNominalRFPower();
        long getTransmitCycle();
        void setTransmitCycle(in long newTransmitCycle);
    };

    interface TestRadio {
        readonly attribute Transmitter transmitter;
        readonly attribute SignalDetector signalDetector;
        float getTxAmplitude();
    };
}
```

![Figure 4.7: OWL generated from TestRadio IDL interface shown in Listing 4.1](image-url)
Operations with multiple parameters

The primary challenge of translating IDL operations that have multiple parameters to OWL is the fact that OWL properties represent binary relations [94] – they link an individual with another individual, or an individual with a data value. The W3C Working Group developed two patterns [94] that may be used for the purpose of representing n-ary relationships in OWL:

1. Introduce a new class that represents the relationship itself, then add n properties with that class as their domain
2. Use lists to represent arguments in a relation

We will explain and show how these patterns can be applied to represent the following IDL operation (defined inside the I interface), which comes from the OSSIE Radio_Control.idl:

```plaintext
void get_gain_range(in unsigned long channel,
                   out float gmin, out float gmax, out float gstep)
```

Representing an n-parameter operation with the use of the first pattern requires creating a class that represents an operation, a class that represents a parameter, an object property that links the operation class with its parameters, and a set of properties that describe the operation (return type and name) and each parameter (position in the operation’s declaration, type and the passing direction). The resulting OWL TBox is shown in Figure 4.8.

When applying this pattern to a concrete operation, a single Operation class must be instantiated and a Parameter individual must be created for each parameter in the operation. Furthermore, each individual parameter must also be linked to individuals representing its type, position and passing direction. The OWL ABox representing the get_gain_range oper-
Figure 4.8: Representing an n-parameter IDL operation in OWL using the class pattern for n-ary relationships

ation is shown in Figure 4.9. Effectively, this pattern leads to an explosion of axioms, which are difficult to write rules for. Note that primitive types of parameters and the return type of the operation must be represented as *individuals.*

The second pattern is more suitable for representing n-parameter operations because it incorporates the order in which parameters are declared. Using this pattern, the operation and each parameter is also represented as a class, but this time parameter classes are organized in a list, by the virtue of two dedicated properties – one pointing to the head of the list, another one to the next element. The pattern also involves adding an additional class to represent the last element in the list, with a restriction of a maximum cardinality of 0 on the property pointing to the next element. The resulting OWL TBox is shown in Figure 4.10. Note that there is no need to represent parameter’s position anymore, since each parameter has already a specific position in the list.
Figure 4.9: Representing the get_gain_range operation from the OSSIE Radio_Control.idl in OWL using the class pattern for \(n\)-ary relationships.

Figure 4.10: Representing an \(n\)-parameter IDL operation in OWL using a list pattern for \(n\)-ary relationships.
Applying the list pattern to the \texttt{get\_gain\_range} operation results in fewer assertions than for the class pattern, however, it still involves a rather large number of assertions. Figure 4.11 shows the resulting ABox (note that the \texttt{@get\_gain\_range\_4} individual is an instance of \texttt{LastParameter} class).

\textbf{Figure 4.11}: Representing the \texttt{get\_gain\_range} operation from the OSSIE Radio\_Control.idl in OWL using the list pattern for \textit{n}-ary relationships.

Although it is possible to represent operations with multiple parameters in OWL, it comes at an expense – it results in a large number of assertions that may seem rather unnatural at times. Most importantly, however, they significantly complicate the automated matching process, because each parameter with the \texttt{out} passing direction in an \textit{n}-parameter operation may correspond to a separate property in the CR ontology. We will elaborate on this issue when discussing the generation of bridge axioms and ABox facts in the following sections.
4.2.6 Generating bridge axioms

Since the process of matching cannot be fully automated, we need to assist it with sufficient information to generate alignments between the CR ontology and the ontologies generated from IDL. For the reasons explained in the previous section, an IDL ontology contains only classes and datatype properties that represent getters and setters in the IDL. Thus the goal of the matching is to create correspondences between these datatype properties and properties in the CR ontology.

When a datatype property in the CR ontology represents a knob or a meter that can be accessed via an attribute or an operation in IDL, we assert a sub-property bridge axiom. For instance, assume that `cr:txAmplitude` is a datatype property representing the value of a transmitting amplitude, and that `idl:Transmitter@setAmplitude` is a datatype property in the IDL ontology representing a setter method that allows for changing the value of the transmitting amplitude. The matching should generate an alignment that can produce the following bridge axiom:

\[ \text{idl:Transmitter@setAmplitude} \sqsubseteq \text{cr:txAmplitude} \]

If this axiom is present in the knowledge base, we can write a rule that will find the equivalent IDL setter method and invoke it, given that some condition is met, as follows:

\[
\begin{align*}
\text{some condition} \\
\text{setter} \sqsubseteq \text{cr:txAmplitude} \\
\text{setter} \sqsubseteq \text{idl:setterMethod} \\
\Rightarrow \\
\text{invoke}(\text{setter}, \text{newValue})
\end{align*}
\]
Before we show how the matching process is assisted, we need to explain why the alignment cannot simply consist of sub-property axioms involving two properties at a time.

**Representing parameters in the CRO ontology**

Our goal is to enable alignment with the Cognitive Radio Ontology (CRO) [95] published by the Wireless Innovation Forum (WINNF). Although this work is still in progress [96], CRO will inevitably become a standard ontology for the CR domain. Our matching process is not tailored for this ontology, it is generic and could work with any other ontology, but we will treat CRO as the target ontology in our examples, because it exhibits some challenges for the matching process.

The CRO ontology borrows fundamental concepts from the DOLCE (Descriptive Ontology for Linguistic and Cognitive Engineering) [97] upper-level ontology. For that reason, every class in CRO is a subclass of one of the following classes: *Object*, *Process*, *Quantity*, *Value*, *UnitOfMeasure*. The top-level classes are shown in Figure 4.12 with a subset of properties defined in CRO.

![Diagram of CRO ontology](image)

**Figure 4.12:** Top-level classes with a subset of properties defined in the CRO ontology [95].
Our focus is on how the knobs and meters are represented in this ontology. Most of the parameters are denoted by \textit{object} properties that link individuals of Object or Process with individuals of Quantity classes or their subclasses. These object properties are sub-properties of either \textit{objectQuantity} or \textit{processQuantity}. Individuals of the Quantity class are linked with individuals of the \textit{Value} class using the \textit{hasValue} object property. Finally, individuals of the Value class are linked with data values using datatype properties like \textit{hasFloat} or \textit{hasInt}, to represent actual values of the parameters.

Consider the fragment of CRO shown in Figure 4.13. Our goal is to assert that a radio participates in a transmitting process with a carrier frequency set to $2.4\text{GHz}$. The radio is represented as an individual of the \textit{Radio} class, subclass of \textit{Object}, \textit{participatesIn} the \textit{Transmitting} process. The \textit{carrierFrequency} \textit{object} property denotes the carrier frequency used in the \textit{Transmitting} process. Its value can be asserted by first \textit{linking} an individual of the Radio class with an individual of the \textit{FloatValue} class using the \textit{participatesIn}, \textit{carrierFrequency} and the \textit{hasValue} object properties. Then, an individual of the \textit{FloatValue} class must be linked to the actual value using the \textit{hasFloat} datatype property. This sequence of assertions is shown in Figure 4.14. Note that for each property in the middle, individuals need to be created in order to form this \textit{chain}. From now on, we will represent such chains of properties using the following notation:

\[
\text{participatesIn} \circ \text{carrierFrequency} \circ \text{hasValue} \circ \text{hasFloat}
\]

Continuing our example, let us assume that there is an IDL ontology with a \textit{Transmitter@carrierFreq} datatype property representing an attribute in the \textit{Transmitter} interface of some SDR, which allows for reading and adjusting the carrier frequency. Our goal is to align this method with the chain of properties representing the value of a carrier frequency in the CRO ontology. Note that we cannot align the IDL operation with just the \textit{carrierFrequency}
Figure 4.13: Fragment of the Cognitive Radio Ontology [95]

property in the CRO ontology, because this would be invalid for two reasons: (1) carrier-Frequency is an object property and Transmitter@carrierFreq is a datatype property, and (2) aligning only to the carrierFrequency property would be insufficient for the Invoker to create a proper ABox. Moreover, we cannot align the IDL operation just to the hasFloat property, because it would be ambiguous – there can be multiple individuals of the FloatValue class, meaning something entirely different than the carrier frequency. Therefore, our goal is to create bridge axiom(s) that would align the sequence of properties in CRO with a datatype property from the IDL ontology, which we denote as follows:

\[
\text{participatesIn} \circ \text{carrierFrequency} \circ \text{hasValue} \circ \text{hasFloat} \equiv \text{Transmitter@carrierFreq}
\]
Using OWL 2 property chain axioms

In order to generate the bridge axiom from the previous section, we need to be able to express chains of properties in OWL. The first version of OWL [46] did not provide any mechanism to express composite properties and it had to be done in rules. OWL 2 [74] introduced a concept of a property chain, which allows for expressing composite relationships by chaining properties together and defining the chain as a subproperty of another property. Note that we cannot say that a property chain is equal to another property.

Probably the most common example of an application of a property chain is the relationship hasUncle, which can only be defined in terms of two properties, arranged in a specific order, for instance:

\[ \text{hasParent} \circ \text{hasBrother} \sqsubseteq \text{hasUncle} \]
Unfortunately, OWL 2 allows for defining property chains using object properties only. This is insufficient for our needs, since every property chain defined in terms of CRO properties must end with a datatype property. For this reason, we extend OWL 2 to support **datatype property chains** – chains that consist of a sequence of object properties terminated by a single datatype property. There are computational reasons why OWL 2 does not include datatype property chains – it can lead to situations where reasoners are forced to do a computation that is undecidable. However, if the reasoner is rule-based, the reasoning will terminate. As a consequence, the datatype property chains may not be supported by some reasoners.

Going back to our carrier frequency example, we can represent the bridge axiom using the newly defined datatype property chain as follows:

\[
\text{participatesIn} \circ \text{carrierFrequency} \circ \text{hasValue} \circ \text{hasFloat} \sqsubseteq \text{Transmitter@carrierFreq}
\]

Given such a bridge axiom in the knowledge base, we can write a rule that will find the equivalent IDL setter method as follows:

\[
some \text{ condition} \\
\text{participatesIn} \circ \text{carrierFrequency} \circ \text{hasValue} \circ \text{hasFloat} \sqsubseteq \text{setter} \\
\text{setter} \sqsubseteq \text{idl:setterMethod} \\
\Rightarrow \\
\text{invoke(setter, newValue)}
\]

**Adding self-restrictions**

Unfortunately, even the datatype property chains can be ambiguous in some cases. Consider the OWL ABox in Figure 4.15, and assume that there are two properties in the IDL ontology
Radio@getDetectorId and Radio@getAmplifierId – that represent operations to access the name of a signal detector and of a power amplifier, respectively. If we construct the datatype property chains and try to align them with the IDL operations, we end up creating the following bridge axioms:

\[
\text{hasSubComponent} \circ \text{componentName} \sqsubseteq \text{Radio@getDetectorId} \\
\text{hasSubComponent} \circ \text{componentName} \sqsubseteq \text{Radio@getAmplifierId}
\]

Unfortunately, both property chains are the same and if we write a rule to find an equivalent IDL method, the reasoner will find both of them. This ambiguity is unacceptable, because the Invoker will not know which method to execute. In order to distinguish these chains we use a similar technique as in [98] and employ another construct that was introduced in OWL 2, called local reflexivity.

Local reflexivity is expressed by defining a self-restriction on an object property and allows for expressing the fact that each individual of the restricted class is related to itself via this property. [98] gives an intuitive example of how using self-restriction differs from

---

**Figure 4.15:** OWL ABox that results in ambiguous datatype property chains
defining an object property as reflexive. Consider the following axiom:

\[
\text{Narcissist} \equiv \text{Person} \sqcap \text{likes some self}
\]

\text{Narcissist} is a class that is equivalent to a set of individuals of class \text{Person} that like themselves. If the property \text{likes} was defined as reflexive, then it would mean that all individuals of class \text{Person} like themselves.

We can use the self-restriction to disambiguate the property chains, by adding \textit{loops}, which help identify the classes on the path and disambiguate the chains. The procedure of adding self-restrictions is represented as an Algorithm 4.2, where \textit{chain} is the original property chain and \textit{path} is the RDF graph consisting of a sequence of classes and properties between the first class and the datatype property in the \textit{chain}.

\begin{algorithm}
\textbf{Algorithm 4.2} Adding self-restrictions to property chains
\begin{algorithmic}[1]
\Procedure{AddSelfRestrictions}{chain,path}
\For {each class on the path}
\State \textit{property} $\leftarrow$ create a functional object property
\State \textit{property.name} $\leftarrow$ \text{is} + class.name
\State \textit{property.domain} $\leftarrow$ class
\State \textit{property.range} $\leftarrow$ class
\State \textit{restriction} $\leftarrow$ add a self-restriction to the class using the property
\State Add \textit{property} to the \textit{chain} in the place where the \textit{class} appears on the \textit{path}
\EndFor
\EndProcedure
\end{algorithmic}
\end{algorithm}

Applying this algorithm to the graph shown in Figure 4.15 results in producing the graph in Figure 4.16. Now the two ambiguous property chains described above would gain a new form, allowing them to be distinct:

\[
\text{hasSubComponent} \circ \text{isSignalDetector} \circ \text{componentName} \sqsubseteq \text{Radio@getDetectorId}
\]
\[
\text{hasSubComponent} \circ \text{isPowerAmplifier} \circ \text{componentName} \sqsubseteq \text{Radio@getAmplifierId}
\]
To make use of chains with self-restrictions, the additional properties must also be included

![Diagram](image)

**Figure 4.16:** RDF graph augmented with loops to disambiguate property chains

in the property chains written within the rules.

**Annotating IDL**

As we explained in Section 4.2.4, the matching process must be assisted to create a full alignment for all getters and setters in the IDL model provided by the SDR. We do so by *explicitly* indicating how the attributes and operations, representing access to parameters in the IDL, correspond to datatype property chains representing these parameters in the CR ontology. The annotations are added in the comments above the IDL constructs. The IDL parser can collect and associate the comment with a particular IDL entity, allowing the matcher to create bridge axioms and generate the Invoker.

The annotations that appear in the comments above attributes and operations provide enough information to create datatype property chains with self-restrictions and are of the following form:

\[
\text{Class1.(objectProperty.Class)}^n\text{.datatypeProperty}
\]

where \(n \geq 0\). Every annotation must begin with a name of the class that is the first item
in the property chain. In the above examples from the CRO ontology, this would always be the Radio class. This name is followed by a sequence of objectProperty.Class pairs, which stand for the next edge and a vertex in the graph leading to the datatype property. The length \( n \) of this sequence depends on the distance between the first class and the datatype property. The last component of the annotation is the name of the datatype property, which points to the actual value of the parameter. The annotation means that this parameter can be accessed or changed with the attribute or operation above which the annotation appears in the IDL model.

When the matcher reads the annotated IDL, for each attribute or operation that is properly annotated, it creates self-restrictions for every class that appears inside the sequence (using the procedure in Algorithm 4.2). Next, it generates bridge axioms using datatype property chains created from the annotation and a property from the IDL ontology that represents the attribute or operation. If the length \( n \) of the sequence is equal to 0, there is no need to create property chains and the matcher can create a simple sub-property axiom.

We illustrate the generation of bridge axioms based on the annotations using an example shown in Listing 4.2. Three constructs are annotated with datatype property chains from the CRO ontology, and two methods are not annotated and will be ignored by the Matcher. Given the example IDL, the Assisted Matcher will generate the following bridge axioms:

\[
\text{participatesIn} \circ \text{is}_\text{Transmitting} \circ \text{transmitCycle} \sqsubseteq \text{Transmitter}@\text{getTransmitCycle}
\]

\[
\text{TestRadio}@\text{radioId} \sqsubseteq \text{componentName}
\]

\[
\text{hasSubComponent} \circ \text{is}_\text{PowerAmplifier} \circ \text{txAmplitude} \sqsubseteq \text{TestRadio}@\text{getTxAmplitude}
\]

In Section 3.2.1 we introduced EDOAL, a language designed for expressing alignments, with the support for complex correspondences. When LiveKB is bootstrapped, the bridge axioms are first expressed in EDOAL, then using Alignment API, EDOAL is translated.
into OWL. Although it may seem to be unnecessary, because Matcher could produce OWL axioms directly, there are benefits of producing EDOAL as an intermediate step. EDOAL is expressed in terms abstracted from the ontology language and it can be used to produce axioms in many languages, e.g. OWL or WSML. This, by extension, provides LiveKB with the opportunity to create axioms in different ontological formats.

The bridge axioms expressed in EDOAL or in OWL tend to be quite verbose. For this reason we included a comprehensive example of all artifacts produced by LiveKB in Appendix C. The code listings include bridge axioms expressed in EDOAL and in OWL.

**Operations with multiple parameters**

In the previous section we showed how $n$-parameter operations can be represented in OWL using two design patterns recommended by W3C to represent $n$-ary relationships [94]. In both cases, the operations in IDL are represented by classes, and their parameters are represented by a number of classes and properties. This representation creates a serious difficulty
for the matching, even if annotations can be provided, since one operation with multiple parameters may correspond to multiple properties in the CR ontology. Consider the method from the OSSIE Radio_Control.idl interface that we used before:

```c
void get_gain_range(in unsigned long channel,
                     out float gmin, out float gmax, out float gstep)
```

The first parameter (channel) is the context information, which must be provided by the CORBA client when the operation is invoked. The remaining parameters represent three distinct values, each of which needs to be represented in OWL as a separate property. However, since the value of each parameter must be associated with a particular context information, every parameter needs to be represented in OWL as an n-ary relationship, using one of the patterns described above. Moreover, if the context information is not known until runtime, the matcher cannot create bridge axioms just by parsing the IDL model.

The `get_gain_range` operation from OSSIE was designed to serve the genetic algorithm (cf. Section 2.3.2), and for each channel the three parameters returned by the radio never change during its operation. If these parameters are well-known, they can be asserted into the ontology manually, once per radio platform, and the matcher would not have to add the bridge axioms, since these values would be already present in the ontology. There are, however, some operations in OSSIE that allow for changing knobs and meters at runtime, which require context information to be provided by the invoker, for instance:

```c
void set_frequency(in unsigned long channel, in float f)
void get_frequency(in unsigned long channel, out float f)
```
These methods can no longer be invoked from within the rules simply as

\[
\text{get(propertyName)} \\
\text{set(propertyName, newValue)}
\]

because they require the context to be provided. It is possible to add new procedural attachments that include context, e.g.:

\[
\text{get(propertyName, context)} \\
\text{set(propertyName, context, newValue)}
\]

but this would not be sufficient in cases when the context is multi-dimensional.

In summary, the LiveKB component currently does not support the \( n \)-parameter operations with context information. As a consequence, these operations must be represented by dedicated procedural attachments. In the future work we will consider replacing the \text{get} and \text{set} methods with ones that follow the \textit{Command} design pattern[36], in which requests are expressed as objects. This would allow for providing an arbitrary number of context parameters from within the rules, while maintaining the generic format of \textit{get} and \textit{set} operations.

### 4.2.7 Generating Invoker

Before the reasoning component can start inference, the knowledge about the current values of radio’s operational parameters must be already present in the KB. Moreover, because of the nature of how the inference engine operates, the \text{get} method returning a value of a single parameter, effectively will \textit{never} be used directly from within the rules (we will explain this later). For these reasons, LiveKB API provides a single \text{getAll} method to fetch values of all parameters at once. This method, realized by the \textit{Invoker} component, returns a set of ABox
axioms that are already expressed in terms of the CR ontology (cf. Figure 4.6), which can be directly asserted into the reasoner’s KB.

In order to realize the getAll method, Invoker needs to know how to access the SDR runtime objects and how to assert values returned by all getters into appropriate ABox axioms. The first part is accomplished by utilizing CORBA Naming Service and by requiring the SDR objects to form a tree-like structure. The second part is realized by combining the object tree structure with the annotations provided for the bridge axioms (explained in the previous section). A detailed description of both parts follows next.

**SDR object tree**

LiveKB, and by extension, Invoker, is a generic component, i.e. it does not assume a concrete API to be implemented by the SDR. In order to automatically invoke getters on multiple objects provided by SDR, Invoker needs to be given some guidelines about how to locate concrete runtime objects. The interfaces in the IDL provided by the SDR are required to form a tree-like structure. The edges in the tree represent object type interfaces or methods returning single instances of IDL interfaces, and vertices represent the instances themselves. The root of the tree should represent the top-level object, from which all other objects can be accessed.

Given such a tree-like structure, the Invoker only needs to know the reference to the top-level object. This part is realized using the Naming Service utility provided by the CORBA middleware. The Naming Service allows CORBA objects to be registered under a user-created name, which then can be used to find the object’s reference. The added benefit of using this service is the fact, that the SDR and LiveKB objects can reside on different processors, even on different machines. This interoperability is provided by ORBs.

In summary, the IDL interfaces provided by the SDR must form a tree-like structure, and the implementation of the root interface in that tree must be registered with CORBA Naming
Service under a name that is provided to the LiveKB at bootstrap. Using reflection, Invoker can identify which interface is implemented by the root object and use this information when building the internal object tree structure. Given the IDL definitions shown in Listing 4.3 and a reference to an implementation of the TestRadio interface, Invoker will create an object tree structure represented in Figure 4.17.

**Listing 4.3** Example of IDL interfaces that form a tree-like structure.

```idl
module api {

    interface SignalDetector {
    }

    interface Power {
    }

    interface Transmitter {
        readonly attribute Power myPower;
    }

    interface TestRadio {
        readonly attribute Transmitter aTransmitter;
        SignalDetector getDetector();
    }
}
```

**Figure 4.17**: Object tree created by Invoker to access SDR’s object in runtime.
If implementations of some interfaces in IDL cannot be reached by invoking a sequence of operations starting from the root object, they are unaccessible by Invoker. This limitation could be easily removed by providing LiveKB with a set of root-level object names that are registered with CORBA Naming Service, and allowing the IDL interfaces to form a forest—a disjoint union of trees. The root element of each tree in the forest would be represented by interfaces, which are implemented by one of the objects registered with the Naming Service. Allowing a forest structure of IDL interfaces would only slightly complicate the implementation, but for the clarity of the explanation, in this work we assume that SDR provides a tree-like structure of interfaces.

**Asserting ABox axioms**

Once Invoker creates the object tree, it can access all getters implemented by the SDR. The values returned by the invocations now need to become part of ABox assertions, which will allow the reasoner to trigger CR rules. Thus, Invoker needs to know which axioms to create and where to put the values. In order to do that, Invoker makes use of the same annotations that are provided in the IDL model to assist the Matcher.

Recall the form in which all IDL annotations appear above attributes or operations that allow for access to particular radio parameters:

\[
\text{Class1.(objectProperty.Class)}^\text{datatypeProperty}
\]

Since every attribute and operation belongs to some interface and every interface implementation must be accessible in the object tree, Invoker has enough information to tie the value returned by the invocation with a set of axioms representing the property chain. In order for the entire set of ABox axioms generated by Invoker to be meaningful, however, all axioms must form a connected RDF graph. As a consequence, Invoker cannot simply create
anonymous individuals for each class in the annotated chain, because this would lead to a disconnected graph, without any value to the rule developer.

For instance, consider the Listing 4.2 and the two RDF graphs in Figure 4.18, which represent different ABoxes that could be generated for the same set of annotations (the values are arbitrarily assigned). The connected graph is of the desired form and allows for referring to one Radio individual with different properties. The disconnected graph, created by generating anonymous individuals (effectively, blank nodes in RDF) does not represent radio’s parameters in a way that it could be used in rules. For example, a rule that is sensitive to a condition including a transmitCycle and a txAmplitude would never trigger, because these properties are not related to the same individual.

The problem of graph connectivity could be solved by assuming that every OWL class
in the annotations has only one member, named using some pattern derived from the class name. However, this would be too limiting for numerous ontologies, and ABox could be created for very specific cases only. This includes the CRO ontology, which is of our particular interest. Since CRO is grounded in the DOLCE upper-level ontology, most values appear at the end of similar sequences, involving individuals of the same classes. Consider the fragment of CRO ABox graph shown in Figure 4.19. Although the individuals FloatValue_1 and FloatValue_2 belong to the same class, FloatValue, they are distinct and cannot be reused.

![Diagram](image-url)

**Figure 4.19:** Example of an ABox graph where multiple individuals of the same class are necessary

One solution to this problem would be to append a number to the class name in the IDL annotations and indicate that for classes with the same suffix number, the same individual should be used in assertions. However, this solution applied to complex ontologies could easily lead to mistakes and result in unwanted RDF graphs. Since in the CRO ABox graphs multiple individuals of the same class are often not reused (e.g. FloatValue), they can be represented using anonymous individuals. For this purpose, we extend the IDL annotations
format to allow for explicitly indicating that an anonymous individual should be created for a given class by appending a * suffix to its name. The new pattern for annotation format looks as follows:

\[ \text{Class1.(objectProperty.Class[^*])}^n \text{.datatypeProperty} \]

The following are some examples of properly constructed annotations:

- \text{Class1.dataProperty}
- \text{Class1.objectProperty1.Class2.datatypeProperty}
- \text{Class1.objectProperty1.Class3[^*].datatypeProperty}
- \text{Class1.objectProperty1.Class4.objectProperty2.Class5[^*].datatypeProperty}

The classes marked with the * prefix will usually indicate the beginning of a branch in the RDF graph that has no sub-branches. For classes without the suffix, Invoker asserts facts with named individuals, assuming each class has one member with a name generated by a pattern. This pattern consists of combining the name of the root interface implementation and the name of the class to which they belong. For instance, if the Radio implementation was available under CORBA Naming Service as radioA, the instance of SignalDetector would be named radioASignalDetector. This allows each radio to create a distinct RDF graph, which is important when radios exchange knowledge and assert ABox facts from the same ontology. If two radios used the same names for their individuals, after asserting both ABox facts into one KB, the two graphs would be merged and misrepresent the reality.

As of now, using the proposed annotation format it is not possible to generate CRO ABox that represents a radio with two complex components of the same type, since this would require creating two distinct and named individuals. This problem is somewhat related
to the problem of operations that require context information, because retrieving values of parameters from two components of the same type requires providing some input that allows selecting a specific component. Augmenting the support for both - multiple parameter operations and more complex RDF graphs - is part of the future work.

**Invoker**

When LiveKB is bootstrapped, Assisted Matcher generates bridge axioms, and an Invoker is created. The process of creating the object tree and asserting entire chains of individuals needs to be done only *once*. Invoker creates a hash table with IDL ontology properties as keys, and special structures as values. These structures hold enough information for Invoker to assert axioms that hold the actual parameter values – a reference to the last individual in the property chain, the name of the datatype property (CR ontology ), a reference to the SDR object and a method that represents a knob or a meter. The references to the SDR objects are found by traversing the object tree. When the reasoner requests the entire ABox, Invoker reuses the RDF graph created at bootstrap and uses the hash table to invoke all the getters and asserts the axioms regarding the datatype properties, effectively adding the leaves in the graph. When the reasoner requests to set a new parameter, Invoker finds an appropriate object and a method in the hash table and invokes the method with the new value.

### 4.3 CRF Interfaces

The CRF architecture makes use of four APIs between its components (Figure 4.20): two specific for the radio domain – *Rx/Tx API* and *DFE API*, one dedicated to controlling the reasoning component – *Reasoner API*, and a generic *LiveKB API* for accessing SDR’s parameters. These APIs turn CRF into a modular architecture that allows for replacement of
its components. Since the use of APIs is of our particular interest in this work, we elaborate on each of the interfaces in the following sections.

**Figure 4.20**: APIs involved in the CRF architecture

4.3.1 **LiveKB API**

One of the major reasons behind designing the CRF architecture was moving away from exploiting radio-specific APIs that are not standardized and may change in the future, and replacing them with one that is more abstract and independent of the radio domain. Such a generic API would allow LiveKB to interface existing software and adapt to changes made in the future. Since both LiveKB and its object factory are available via CORBA, we present this API in the form of IDL that they implement (Listing 4.4).

Before LiveKB can be used, it must be instantiated by the LiveKBFactory. Reference to the LiveKBFactory implementation can be found in the CORBA Naming Service using a name specified in the LiveKB configuration file. Once a reference to the object factory is obtained, it can be used to create an instance of LiveKB by providing it with the annotated IDL model, the name of the SDR’s root object, and the CR ontology.
Listing 4.4  LiveKB API described in IDL

```idl
module livekb {

  exception LiveKBException {
    string errorMessage;
  };

  interface LiveKB {
    string getAll()
    raises (LiveKBException);

    any get(in string property)
    raises (LiveKBException);

    void set(in string property, in any value)
    raises (LiveKBException);
  };

  interface LiveKBFactory {
    LiveKB getInstance(in string model, in string rootName,
      in string ontology)
    raises (LiveKBException);
  };
}
```

LiveKB offers three completely generic operations, `getAll`, `get`, `set`. While the first one invokes all getters in the IDL to create an ABox representation in the domain ontology, the second and third methods allow for getting or setting values of parameters available via IDL. As an argument, they take a name of a property in the IDL ontology, which is found by the reasoner from within the rules (details are explained in the preceding section). Each operation may throw a custom `LiveKBException`. This exception can be thrown for several reasons, e.g. SDR objects are no longer available or the name of the property parameter does not correspond to any property in the IDL ontology.

The names of the properties used as arguments are found by the reasoner using the bridge axioms. The rule writer needs only to refer to terms from the CR ontology. This API does
not contain any SDR-specific information. In fact, it does not contain terms specific to any domain. LiveKB API is more abstract than traditional APIs because it does not constrain the reasoner to a fixed number of radio-specific parameters selected during the design time. This feature of the CRF architecture allows the rules to be fully reusable. The hard-coded part of the interface is oblivious to the name of parameters or methods used to access them. The radio-specific information, instead, is used when creating the LiveKB component.

4.3.2 Message Layer APIs

The architecture of CRF originated from the OBR architecture, where the Data Out and Data In components are supposed to send and receive messages from the DL layer [56]. This was never implemented, instead, in the prototype implementation the OBR directly communicated with the radio’s physical layer by invoking native methods via JNI.

The Rx/Tx API was not the main focus of this work, and it may be replaced in the future. Currently, for the SDR to be compatible with CRF, it must be available via CORBA and implement the simple API shown in Listing 4.5. This API allows the DIO component

Listing 4.5  Rx/Tx API required to be implemented by the SDR

```plaintext
module sdr {

    interface SDRListener {
        void rx(in string msg, in boolean ok);
    };

    interface SDR {
        void tx(in string msg);
        void addRxListener(in SDRListener cc);
        void removeRxListener(in SDRListener cc);
    };
}
```

to send messages using the tx method and receive messages by implementing a callback rx
method. Note the second argument of the \texttt{rx} method – this boolean value indicates whether the received message was corrupted or not. If CRF was connected with SDR at the layer that already offers a reliable transmission, this parameter would become obsolete. The SDR must implement the \texttt{tx} method and the two methods that allow DIO to (de)register as a listener.

If the SDR provides unreliable transmission, the DIO component must implement a reliable data transfer to support exchange of control messages. Moreover, DIO filters out incoming control messages and passes them to the cognitive layer. The remaining data messages are forwarded to the DFE component, available via CORBA for the upper layer. DFE provides the API shown in Listing 4.6. Note that this API is almost identical to the Rx/Tx API, with the difference that the \texttt{rx} method has only one parameter – there is no need to inform the upper layer of corrupted packages, since DIO guarantees reliable transmission.

The current Message Layer APIs are rather trivial and they should be easily implemented by neighboring components - SDR and the upper layer applications. The transfer of control messages between DIO and cognitive layer, and the transfer of data messages between DIO and DFE, must be implemented using buffers, which allow asynchronous access. These buffers are necessary to compensate for situations in which messages on either end are not

\begin{lstlisting}[language=IDL]
module idl {
  interface DataListener {
    void rx(in string msg);
  }

  interface DataFrontEnd {
    void tx(in string msg);
    void addDataListener(in DataListener cc);
    void removeDataListener(in DataListener cc);
  }
}
\end{lstlisting}

Listing 4.6  Data Front-End API provided for upper-layer applications to send and receive data.
consumed fast enough. The outgoing messages from DIO to SDR are pooled together in one buffer. Ideally, the buffers should support message priority and allow the control messages to be handled as soon as possible. This would help avoid situations in which radios cannot collaboratively improve link quality, because control messages are stuck in the buffers, while data messages are being retransmitted by the RDT.

### 4.3.3 Reasoner API

In this section we discuss various possibilities for a reasoner’s API. This discussion is provided only for informational purposes since the design of Reasoner API is not the focus of this research.

Ideally, the interface between Monitor Service and the reasoner would be based on a standard API, allowing different reasoners to be used within the CRF architecture. Moreover, such an API would introduce a layer between the two components, which would abstract away the details of how the inference engine’s KB is designed and organized. Since we chose OWL as the language for knowledge representation, we focus on an interface to a Description Logic (DL) reasoner. Several APIs already exist, however, none of them has been universally adopted:

- Description Logics Implementation Group (DIG) interface [99] provides means to access a DL reasoner in an implementation-neutral fashion. The reasoner is required to respond to XML messages sent over HTTP connections. Essentially, the interface defines the XML schema for this communication. Although DIG is used by a very popular ontology editor, Protege\(^1\), due to its complexity, it has been implemented by a very limited number of reasoners. There is an ongoing effort to develop a successor of DIG\(^2\), however, it is *not* going to be backwards compatible.

\(^1\)Protege editor is available at [http://protege.stanford.edu/](http://protege.stanford.edu/)
\(^2\)DIG 2.0 resources are available at [http://dig.cs.manchester.ac.uk](http://dig.cs.manchester.ac.uk)
• OWL API [100] is a Java interface for managing OWL ontologies. It is thus platform-dependent and narrower API than DIG, because it only allows for manipulation of OWL. This API is being actively developed and implemented by leading reasoners. It also supports OWL2, a new version of OWL.

• Storage and Inference Layer (SAIL) is part of a Sesame\(^3\) open source framework for storage and querying of RDF data. It is available in Java and has a strong support for query languages. SAIL was updated in 2008 to a version 2.x, losing backwards compatibility with its predecessor.

• Jena\(^4\) is a Sesame competitor, also implemented in Java, offering support for different query languages. Unlike Sesame, it not only provides an API for RDF, but also for OWL. Jena uses a very succinct format for writing rules, which is very easy to read and write by humans, but is hard to manipulate (because it is not XML-based) [101].

• API4KB\(^5\) is an API that is currently being developed by the OMG group, which aims at standardizing it and providing support for both RDF and OWL tools.

Existing APIs for accessing reasoner functionality differ significantly: they support different query languages and data formats. Each API is designed with a different philosophy in mind and it requires a fair amount of effort to add support for a new API to an existing piece of software. Moreover, leading reasoners support different interfaces, not to mention different versions of the same API are sometimes incompatible, making the task of choosing the right one even more difficult. The choice of this API remains part of the future work. In current implementation we designed a simple, custom Java API.

\(^3\)Sesame is available at http://www.openrdf.org

\(^4\)Jena is available at http://jena.sourceforge.net/

\(^5\)Request for proposal is available for the OMG members at http://www.omg.org/cgi-bin/doc?ontology/10-03-01.pdf
4.4 Choice of middleware

The CRF architecture extensively makes use of the CORBA middleware — Object Request Broker (ORB) and services — in order to provide platform independence between the LiveKB component and the SDR. Specifically, CORBA allows LiveKB to invoke the SDR API methods corresponding to radio’s knobs and meters and the CORBA Naming Service allows the LiveKB to locate the SDR object references dynamically. Most importantly, however, the LiveKB design heavily depends on IDL and its annotations embedded in IDL comments.

Using CORBA as the middleware for the radio may seem like a heavy-duty solution, since incorporating CORBA does introduce some overhead. A benchmark [102] comparing the performance of a GNU Radio and OSSIE based waveforms showed that use of CORBA significantly increased the CPU usage and introduced a major latency in the inter-component communication. Perhaps CORBA could be replaced with a different mechanism that allows object lookup and control in a language-neutral fashion. However, it seems that the performance of CORBA could be greatly improved if only a more efficient transport protocol than TCP/IP was used, instead [103]. Moreover, CORBA is a widely-accepted standard in the wireless community: the CORBA-intensive SCA was a great success, and there are ongoing efforts to implement CORBA on both DSPs and FPGAs [103], further improving the performance of CORBA-based SDRs.

An alternative to using CORBA would be to follow the approach taken in the architectures from Virginia Tech, where XML plays the role of the interface language. However, as [104] indicates, utilizing XML for the middleware layer should only be used when the messages exchanged between components are at some point intended for human consumption. Most importantly, relying on simple XML messages would significantly impact the design of LiveKB, possibly prohibiting it from performing reflective invocations that are possible with CORBA due to the existence of stubs and skeletons.
4.4.1 Web Services

An alternative to using CORBA in the CRF would be to introduce more technology-neutral Web Services. The World Wide Web Consortium (W3C) defines [105] the Web Services as follows:

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Web services bridge the boundaries between the enterprises, and those between different middleware platforms within the enterprise itself [106]. They rely as much as possible on open, XML-based web standards and are especially suitable for client/server systems with modest performance requirements [107]. There are some similarities between the two technologies:

- Web Services are described with WSDL, CORBA components are described with IDL. While WSDL is message-oriented, focusing on the content of messages passed between client and server business objects, IDL is object-oriented, focusing on fine-grained interactions between client and server programming language objects.
- They both use a language-neutral wire protocol [108]. CORBA defines a binary GIOP protocol and ORBs usually use the IIOP, GIOP implementation over TCP/IP network layer. ORBs may also be implemented to use different transfer protocols, however, they require a substantial effort to implement, because ORBs must adhere to CORBA standards. Web Services, although traditionally implemented with the use of XML-based SOAP messages transferred over HTTP, are independent of any particular transport
protocol. For instance, a Web Service can be specified to use REST[109] for its transport protocol, which does not make use of XML at all.

- Both CORBA and Web Services provide means for object/service lookup. While CORBA provides a Naming Service to find object references, Web Service clients may find service references using the Universal Description, Discovery and Integration (UDDI) servers. Web Services are more business-driven and the UDDI can be thought of as an advertisement space for businesses to publish their services.

4.4.2 Web Services and the CRF

The wireless community has been extensively utilizing CORBA since the SCA-based SDRs were introduced. The focus of the CRF architecture is the interface between the reasoner and a variety of different SDRs. Given that many SDRs are already implemented using CORBA, this middleware platform was a natural choice for us. On the other hand, it is a common practice to wrap CORBA resources with Web Service interfaces — OMG defines a CORBA to WSDL/SOAP\textsuperscript{6} and a WSDL/SOAP to CORBA\textsuperscript{7} mapping.

Using Web Service based APIs for interfacing SDRs would provide a more universal mechanism for using SDRs in the distributed computing environment, e.g., establishing ad-hoc networks, peer-to-peer interactions, or computing in the cloud. Web Services could even be considered as a replacement for CORBA. However, at this moment there are no known implementations of SDRs that make use of Web Services. Consequently, moving in this direction would require significant efforts to promote such an approach among the SDR vendors. One of the intended objectives of this thesis is to convey the message to the SDR community about the use of reasoners. A more ambitious goal of promoting the use of Web Services is beyond the scope of this work.

\textsuperscript{6}http://www.omg.org/spec/C2WSDL/1.2.1
\textsuperscript{7}http://www.omg.org/spec/WSDL2C
Some of the concerns related to the use of Web Services for interfaces among SDR components are:

- The SCA-based radios would need to incorporate two middleware platforms, further increasing the demand for resources.
- In addition to a Web Service based interface, the SDRs still need to provide a platform independent Rx/Tx API which needs to enable binary data transmission. Replacing this interface by a Web Service based interface could lead to a less efficient implementation (e.g., more power consumption, decrease in speed), especially if the Web Services used SOAP for its transport mechanism, which requires XML (de)serialization.
- It should be noted, that currently many SDR vendors, especially those who manufacture hand-held devices, are opposing even the use of CORBA claiming that it adds too much complexity to such devices.

### 4.5 Dynamic change of the knowledge base

Changing the values of the SDR’s parameters at runtime requires frequent updates to the KB, which poses a challenge for the OWL-based inference, specifically because OWL is monotonic and in monotonic logics facts can only be added, but not updated. This is due to the fact that OWL is based on the open world assumption model, where only facts whose truth values are unknown can be modified. Once proven to be true or false, facts retain their truth value regardless of new facts that might become part of an existing KB [67]. This feature of the OWL language becomes particularly problematic when dealing with retraction, i.e. removing facts from the KB, because it is not clear which other facts should be removed as well, in order to maintain consistency. Efforts [110] to support OWL-based reasoning with retractable inference lead to constraining OWL to its least expressive sublanguage, called OWL Lite, and even moving outside of the OWL semantics by introducing closed world
The problem of monotonicity of OWL may be approached in several ways:

- Restart the reasoner with a new ABox each time a change occurs. It is possible that this solution may prove to be inefficient in the real world applications, resulting, for instance, in a shorter battery life of the radio due to more extensive use of its resources.

- Retain old facts and mark new ones with time stamps. This severely complicates developing rules, because the KB may include multiple facts about the same object, each with a different time stamp.

- Use a reasoning engine, which supports retraction. This approach would tie the framework to a particular reasoner, because retraction is not a standard operation and reasoners offer different types of retraction, or none at all. Moreover, it is not included in any of the existing reasoner APIs, although there is an effort to include it in the next version of the DIG interface [111].

Although using retraction may drastically improve performance in situations when only a fraction of the ontology changes each time [111], due to the problems mentioned above, current implementation of the CRF architecture follows the first approach and restarts the reasoner each time a change occurs.

### 4.5.1 Rules and events

The knowledge base of the reasoner must be updated each time the radio’s parameters change, so that the radio is aware of its current state. There are essentially two ways in which the reasoner can be informed that a change occurred: by a push notification from the radio or based on a periodic pull, initiated by the reasoner itself. The first scheme requires that the radio implements some API, which allows for registering listeners. Each
time an event occurs, the radio invokes some specific method for each registered listener. The second option is to periodically pull all information from the radio and check if anything has changed. This periodic pull may lead to two issues: missing important events that happen between the times the pull occurs, or unnecessary utilization of radio’s resources when no events occur at all.

Current implementation of the CRF architecture uses the pull scheme. Each time a certain fixed amount of time has passed, and if the FIPA FSM is in the IDLE state, it creates an initEvent, which transitions FIPA FSM to the state in which the reasoner is invoked with current ABox values. Because of the declarative nature of the rules, the LiveKB component fetches current values of all parameters, even if only a few or none have been updated since the last time the reasoner was run. The implementation should allow to disable this self-trigger and let the radio react only to incoming messages when in the IDLE state.

LiveKB does offer the get method, which can be used to fetch values of specific parameters only. However, it would only make sense to use this method if (1) the reasoner supports retraction and if (2) the SDR implements the push notification. It is more likely that the get method will be used for other purposes. For instance, FIPA FSM could use it to check the value of some parameter and depending on its value, trigger the reasoner.

4.6 Behavioral description

Since both the reasoner and the LiveKB component are domain-independent, they need to be configured with the domain-specific information before they can be used. Once configured, the reasoner can process the CR rules and control the radio’s behavior accordingly. To better understand this process, we describe the initialization and the runtime stages in more detail.
4.6.1 Initialization

During the initialization, the LiveKB component must be instantiated by the LiveKBObjectFactory by providing it with the annotated IDL model of the SDR, the name of the SDR’s root object registered in CORBA Naming Service and the CR ontology. Before the object factory returns the instance, it bootstraps LiveKB. During its bootstrap, LiveKB parses the annotated IDL model, generates the IDL ontology, the bridge axioms, the ABox graph without the leaves (representing actual values), and an object tree. Then using reflection it traverses the object tree to find references to the objects and the methods that are stored in a hash table indexed by the names of the corresponding IDL ontology properties.

Once an instance of LiveKB is created, it is passed to MS, which uses LiveKB to fetch ABox before it starts the reasoner. To instantiate the reasoning component it needs to be supplied with the CR ontology, rules and the reference to the LiveKB instance. The reasoner needs LiveKB to instantiate procedural attachments get and set, which can later be used from within the rules.

Once MS and the reasoner are created, DIO and DFE are instantiated as well and the CRF is ready to respond to incoming messages. Figure 4.21 shows the sequence diagram depicting the most important parts of the initialization stage.

4.6.2 Runtime

During runtime, DIO intercepts control messages that come from other radios and passes them to MS. MS parses incoming control messages to create objects representing ACL communicative acts, which are passed to the implementation of a FIPA finite state machine (FSM). Depending on the current state and the type of an ACL message, FIPA FSM may generate a failure and return it back to DIO without even invoking the reasoner. If, however, the current state can accept the ACL message, FSM may transition to a new state and run
the reasoner passing it the contents of the message. Before the reasoner is started, LiveKB is invoked to fetch the current ABox values and together with the IDL ontology, the bridge axioms and the incoming message, it is asserted to reasoner’s KB and the inference is started. While the reasoner is executing, some rules may trigger and cause the use of LiveKB’s set methods. If that is the case, LiveKB uses its internal hash table to fetch a reference to the SDR’s particular object and a method corresponding to the requested knob change. The method is invoked using reflection. As a result of inference, by the virtue of procedural attachments, the reasoner may generate an ACL message that should be handled by FSM. If the produced message is acceptable in the current state, it is passed to the DIO, which transmits the message over the air.

This scenario, shown as a UML sequence diagram in Figure 4.22, is only one example...
of how the incoming messages may be handled. The actual behavior is driven by the mix of FIPA FSM and rules. While FIPA FSM implements standard and generic interaction protocols, rules are written for specific use cases and can be edited without modifying the hard-coded implementation.

Figure 4.22: Sequence diagram showing one scenario of handling incoming control message

4.7 Roles and artifacts

As mentioned above, because the reasoner and the LiveKB component are domain-independent, they need to be configured with the radio-specific information during the initialization stage. That information is provided by the domain experts and by the SDR vendors. The CRF architecture provides both parties with a reasonable separation of concerns. Figure 4.23 shows
both roles and artifacts that they are responsible for producing. Both domain experts and the SDR vendors are responsible for delivering radio-specific artifacts, required to configure the general-purpose reasoner and the LiveKB components. The annotations in the SDR IDL model bind the software methods with the CR ontology.

![Figure 4.23: Division of roles under CRF architecture](image)

Domain experts express the knowledge related to the cognitive radio in two forms:

- **CR ontology**, which includes the general knowledge of the radio domain.
- **CR Rules**, which describe the cognitive behavior of the radio in declarative form, using terms defined in the ontology.

In order to make their products compatible with the CRF architecture, SDR vendors are responsible for:

- **Annotated IDL models** respecting the constraints put forth by LiveKB
- **Making the SDR available via CORBA**

The artifact that binds the efforts from both roles are the IDL annotations, which map the SDR’s model to the CR ontology. This mapping allows the LiveKB component to find parameters within the radio structure that correspond to abstract terms from the reasoner’s
knowledge base. The mapping needs to be updated upon changes to the ontology or to the software model. In order to decrease the amount of necessary maintenance of this mapping, we recommend using the standardized CR ontology [95]. This should give the SDR vendors the freedom to develop their radios according to their needs, yet still making it possible for their radios to be used in the CRF framework.

4.8 Implementation

At the present time a functional prototype of CRF has been implemented. Appendix A shows the details of how the FIPA state machine was designed and which interaction protocols were supported. Appendix C shows all the artifacts produced by the current implementation of LiveKB. The following sections provide a little more detail about the choices made during the implementation of the architecture.

4.8.1 Implementation language

Since LiveKB is required by CRF to be used via CORBA, it can be implemented in any language that has support for this middleware and for reflection. The same applies to the software part of the SDR. We implemented every component in Java, although due to the use of CORBA, this choice has no effect on the platform dependence of the entire architecture. We have successfully interfaced CRF with a radio, for which the software part was written in Python.

4.8.2 Reasoning component

The implementation used only one reasoner, BaseVISor [112]. It is a forward-chaining inference engine based on a Rete network, optimized for the processing of RDF triples. It is developed by Vistology, Inc. and is available for free for research purposes. BaseVISor is
written in Java and can perform inference with a subset of OWL 2 axioms and over rules, which suited our needs. In addition, because BaseVISor is rule-based, it supports reasoning with datatype property chains, which is crucial for processing the bridge axioms generated by LiveKB. Furthermore, since MS and FIPA FSM were implemented in Java, interfacing BaseVISor was very straightforward.

4.8.3 Reasoner API

The choice of the best reasoner API was not the main focus of this work. Instead of making this decision and devoting significant amount of time to utilize a particular API, we designed a simple interface, limited to a few application-specific methods, shown in Listing 4.7. In the future, we may require every component to be available via CORBA and update this interface to IDL, or replace it by one of the APIs that are already available (cf. Section 4.3.3).

Listing 4.7 Java API implemented by the reasoning component

```java
package edu.neu.ece.crf.reasoner;

public interface ReasoningComponent {

    public void configure(String ontology, String rules, LiveKB livekb);
    public Collection<ACLMessage> run(String msgContent);
    public String runQuery(String query);
    public void resetKB();
}
```

The first method allows MS to provide the domain ontology, rules and the reference to the LiveKB component during the initialization stage. The remaining methods are invoked from within the implementation of FIPA FSM. While `run` can be used to start inference, `runQuery` to execute a query, the `resetKB` is invoked each time the state machine transitions back to the IDLE state. This allows the reasoner to forget facts that came from other radios and may no longer hold.
4.8.4 LiveKB component

One of the reasons why LiveKB was implemented in Java was the fact that this language offers a solid reflection mechanism, which is crucial for a successful implementation of LiveKB. During bootstrap, LiveKB generates Java stubs from the IDL description using an external tool. Once generated, stubs are dynamically compiled and loaded into the Java Virtual Machine (JVM). The last step is necessary for the reflection mechanism to work, because it requires class descriptions to be loaded in the same JVM where LiveKB is located. Finally, the root object reference is retrieved from the Naming Service.

4.8.5 Bridge axioms

In the current implementation, when LiveKB is bootstrapped and the IDL is parsed to create bridge axioms using the annotations, LiveKB first generates an EDOAL alignment document and then uses Alignment API [73] to generate OWL axioms. However, since the Alignment API does not support the generation of datatype property chains, we modified its implementation to allow matching datatype property chains with datatype properties. This implementation could entirely avoid EDOAL, and directly produce OWL axioms. However, one might wish to edit the EDOAL document manually and add more correspondences after LiveKB produced it. EDOAL could be a starting point for matching $n$-parameter operations.

4.8.6 Reliable Data Transfer (RDT)

The CRF architecture incorporates FIPA interaction protocols to allow different radios to exchange knowledge among each other. As explained in Section 3.3, these protocols assume that agents (radios) can exchange data in a reliable fashion. In order to meet this requirement we decided to incorporate the reliable communication functionality into the CRF implementation, and thus remove the burden of requiring this feature from the underlying SDR. The
problem of RDT has been well-known in the networking community for years, where several protocols to serve this purpose have been designed. Most notably, the Transmission Control Protocol (TCP), one of Internet’s cornerstones, provides the data transfer reliability to upper layers. The following information is largely based on [113].

The problem of RDT it is usually associated with the transport layer in the OSI network stack, although it occurs at the link and application layers as well. In our case, we required RDT at the link layer (which is where the Monitor Service operates) on top of the unreliable PHY layer (implemented by SDRs). Protocols that provide RDT create an abstraction of a perfectly reliable channel, even though they may operate in an unreliable environment, which is exactly what we needed. RDT means that transferred data is not corrupted or lost and that it is delivered in the order in which it was transmitted.

RDT protocols used in networking belong to a group of *Automatic Repeat reQuest* (ARQ) protocols, which are based on retransmission of lost and corrupted packets. The transmitter knows that it needs to retransmit a packet because it either did not receive a positive acknowledgement (ACK) on time, or it did receive a negative acknowledgement (NAK) from the receiver. The retransmissions and the addition of ACK and NAK control packets ultimately decrease the channel bandwidth, thus the objective is to minimize the overhead created by the RDT requirement.

Out of the three types of ARQ protocols – Stop-and-wait, Go-Back-N and Selective Repeat (SR) – described in [40], we chose to implement SR, the most sophisticated of the three, but also one with the best overall performance. SR is a pipelined protocol, which uses the abstraction of a sliding window and allows for out-of-order acknowledgements by means of buffering. We implemented this protocol in Java, although it communicates with the PHY layer using CORBA, so this choice has no influence on the platform independence of the entire architecture. Moreover, the implementation allows for disabling SR on demand – via GUI, or a configuration file – which may be desired if the SDR already provides RDT.
One of the current limitations is that our SR implementation assumes a communication between two nodes only and it would have to be extended to support multiple connections at once. This would require an addition of a radio ID in each packet, so that the sender could be identified at the receiving end.
Chapter 5

Evaluation

We begin this chapter by discussing the demonstration that made use of CRF to showcase a successful implementation of a knowledge-based CR. Then we evaluate the CRF architecture in two ways. First, we review the architecture against the requirements specified in Section 2.2. Then we evaluate the costs and benefits of using LiveKB as opposed to using domain-specific APIs. We focus on LiveKB since this is the central part that provides CRF with domain adaptability.

5.1 CRF demonstration

Using the implementation of CRF described in Chapter 4, we were able to use the CRO ontology and some rules to realize a specific scenario. We showcased a demo of our implementation during the Product Exhibition at the SDR’10 Technical Conference and received a very positive feedback from the community.

In this section we provide some more detail about the SDR platform that we used and the scenario that was implemented. We conclude this section with some observations about the performance of CRF.
5.1.1 SDR platform

For the demo, we used USRP\textsuperscript{1} and GNU Radio\textsuperscript{2} as our SDR platform. A brief description of each of them follows next.

Software – the GNU Radio toolkit

GNU Radio is an open source software development toolkit that provides signal processing runtime and processing blocks to implement software radios using low-cost RF hardware and commodity processors [3]. It has a large and active user base that spans across hobbyist, academic and commercial environments.

The toolkit offers a Python\textsuperscript{3} interface to implement waveform applications, although the performance-critical signal processing part is implemented in C++ using the processor floating point extensions, where available. Because Python is a dynamic language, it allows for rapid application development. The GNU Radio provides the Python applications with a data flow abstraction — signal processing blocks and connections between them. Each block has a number of input and output ports, and each port has an associated data type. Developers can extend the set of available signal processing blocks by implementing them in C++ and then wrapping them with the Simplified Wrapper and Interface Generator\textsuperscript{4} (SWIG), which allows them to be used from within the Python waveform applications.

GNU Radio has been successfully used on a range of operating systems: Linux, Windows (via Cygwin), Mac OS X, FreeBSD and NetBSD. The toolkit also comes with a GNU Radio Companion (GRC), a GUI for building waveforms using a set of standard processing blocks. The GRC generates Python code, which for simple applications can be executed without any modification.

\textsuperscript{1}The product is sold by Ettus Research \url{http://www.ettus.com/}.
\textsuperscript{2}The project is hosted at \url{http://gnuradio.org}
\textsuperscript{3}Available at \url{http://python.org}
\textsuperscript{4}The project is hosted at \url{http://www.swig.org}
Hardware – USRP

The Universal Software Radio Peripheral (USRP) belongs to a family of products that allows for creating a high bandwidth SDR with a use of a general purpose computer. The high-speed operations like the digital to analog and analog to digital conversion, decimation and interpolation are done on the FPGA. The basic setup requires a computer with a USB 2.0 port, an antenna and at least one daughterboard for a particular frequency range. The available daughterboards cover ranges from DC to 5.9 GHz.

GNU Radio was originally developed to provide a high level toolkit for building waveform applications to be used solely with the USRP. However, the toolkit can be used to interface different hardware, and the USRP can be used without the GNU Radio, via dedicated drivers.

5.1.2 Implemented Scenario

The scenario that was implemented and showcased utilized two radios, which collaboratively performed link optimization [114]. In order to create some data traffic between the radios, we connected image transceiver applications to the Data Front-End on both platforms. While one radio was sending an image to another, the receiver tried to optimize power efficiency of the sender. It was doing so by sending requests to decrease or increase the transmitting amplitude (knob) based on the value of the perceived mean signal-to-noise ratio (meter).

Figure 5.1 shows a screenshot of the GUI that was developed for the demonstration. The snapshot shows the receiver’s perspective – its measured mSNR and the sender’s power efficiency (objective function) are plotted together. The shaded region indicates acceptable values of mSNR. When the current mSNR was outside the region (signal was too strong, or too weak), the receiver was first sending a query to get the value of the sender’s transmitting amplitude, then sending a request to adjust it. Eventually, the radios would negotiate a value of the sender’s transmit amplitude that resulted in a satisfactory value of the mSNR.
on the receiver’s side. In order to continue the demonstration we were bringing the radios closer or farther from each other, which resulted in a continuous control message exchange.

Both radios were running the same GNU Radio application and provided the same IDL to LiveKB, although they were running on different operating systems. The internal buffers of CRF were implemented on top of ActiveMQ\(^5\).

\(^5\text{ActiveMQ is an open-source project developed by the Apache Software Foundation at } \text{http://activemq.apache.org/} \)
5.1.3 Observations

While implementing the rules it was not always clear how to best coordinate the FIPA state machine with the reasoner and its KB. This is certainly an opportunity for future research. On the other hand, having the support from LiveKB made it very convenient to write rules and specify queries without focusing on the radio interfaces. The rules were written exclusively in terms of the CR ontology. They were developed independently from the radio interfaces by different people.

During the demonstration, it turned out that the image sending application was probably too aggressive and the CRF’s buffers would often get filled up, stalling the upper-layer application. At the time, ActiveMQ had very limited support for message priority mechanisms and the control messages, despite higher priority than data, would often have to wait a while, before they were sent out. This resulted in longer response times, although the execution was reliable. It was a simple matter to resolve this issue by modifying the rules.

5.2 CRF and the requirements for a CR architecture

In this section we describe how the proposed CRF architecture fulfills the requirements that were specified in Section 2.2. Note that none of the requirements are necessarily specific do the radio domain; even the SDR-independence really means independence from a specific API.

5.2.1 Reusable knowledge

Support for reusable knowledge requires that the same rules and ontology can be executed on different radios and provide the intended results. To test this feature, the rules must require that some radio parameters (meters) be read and some parameters (knobs) be modified at run time. Also, the radios should differ in the IDL that they provide.
To show CRF’s support for knowledge-reusability, we developed two different radio interfaces, one in Python (Listing 5.1) and one in Java (Listing 5.2). They both provided annotated IDL models with annotations to the same properties in the CRO ontology. They differed in the structure: The **PythonRadio** IDL comprised several interfaces, but made use only of attributes. The **AvantRadio** included only one interface and made use only of operations (with different passing directions, i.e., the “in” vs. the “out” variable types). The different structure resulted in a different object tree that LiveKB had to traverse to find object and method references.

**Listing 5.1  IDL model implemented by a Python application**

```java
module api {

    interface SignalDetector {
        // Radio.hasSubComponent.SignalDetector.signalToNoiseRatio.
        // Decibel*.hasValue.FloatValue*.hasFloat
        readonly attribute float signalToNoiseRatio;
    }

    interface Amplifier {
        // Radio.hasSubComponent.PowerAmplifier.txAmplitude
        attribute float txAmplitude;
    }

    interface PythonRadio {
        // Radio.componentName
        readonly attribute string myID;
        readonly attribute SignalDetector mySignalDetector;
        readonly attribute Amplifier myAmplifier;
    }
}
```

To show that the same set of rules and ontology can be executed on different radios, we executed the rules that were used in the SDR’10 demonstration (described earlier). Specifically, we ran the same scenarios twice, by allowing both radios to be the receiver and the
Listing 5.2  IDL model implemented by a Java application

```java
module api {
    interface AvantRadio {
        // Radio.componentName
        void get_ID(out string id);
    }

    // Radio.hasSubComponent.PowerAmplifier.txAmplitude
    void get_TxAmplitude(out float txAmp);

    // Radio.hasSubComponent.PowerAmplifier.txAmplitude
    void set_TxAmplitude(in float newTxAmp);

    // Radio.hasSubComponent.SignalDetector.signalToNoiseRatio.
    // Decibel*.hasValue.FloatValue*.hasFloat
    void get_SNR(out float snr);
}
```

sender, i.e. both radios had to change their parameters at some point. The radios successfully exchanged knowledge and transmitters fulfilled receivers’ requests to change local parameters. The requests generated and sent to the transmitters were in the form of rules, as shown in Listing 5.3. The new value, marked in the code as @1, was injected by a procedural attachment.

Even though the two radios implemented very different IDL models, they were able to execute the same rules and use the same ontology. Moreover, since the rules were written only in reference to the common ontology, they could be exchanged by the radios as FIPA requests.

In summary, as long as radios implement IDL models that can be processed by LiveKB, knowledge-reusability is fully supported by CRF. Not only rules can be written solely in ontological terms, they can also be exchanged and invoked on other radios that implement CRF.

134
5.2.2 SDR-independence

The objective of this step is to identify the conditions under which LiveKB can access an SDR’s knobs and meters. As a result of this investigation, we identified the following list of requirements:

1. SDR parameters are accessible via CORBA
2. The runtime objects form a tree-like structure and the reference to the root is available via CORBA Naming Service
3. SDR implements IDL that respects the following constraints:

(a) Operations that are used to access knobs and meters have either (1) no parameters, or (2) one \texttt{in} parameter, or (3) one \texttt{out} parameter. In case (1), the return type is primitive. In cases (2) and (3), the return type is \texttt{void}.

(b) Operations that are used to access knobs and meters are properly annotated, i.e. annotations are of the form \texttt{Class1.(objectProperty.Class[*])\texttt{n}.datatypeProperty}.

(c) Annotations combined together form a “proper” ABox, i.e. they form an RDF graph that is expected by the rule writer (see 4.2.7).

Given such constraints, CRF may have no support for some legacy SDRs, however, if an SDR is already available via CORBA and it follows the above requirements, it can be used by CRF out-of-the box. Note that CRF can interface numerous SDRs without requiring them to implement a specific API and without the need to implement an interface-dedicated code. Moreover, when radios change their APIs over time, as long as they still support the above requirements, they can be interfaced by LiveKB, even if the new version is not backwards-compatible.

The radios used in the SDR’10 demo and in our other experiments, all provided different APIs, although they fulfilled the LiveKB requirements. In all cases, the reasoner was able to successfully access the knobs and meters and to fulfill the expectations of the scenarios. In summary, the support for SDR-independence in CRF is partial, because it requires some architectural constraints to be respected.

5.2.3 Platform-independence

This requirement is supported in CRF by incorporating the CORBA middleware, i.e., this support relies on the scope of platform independence provided by CORBA. LiveKB can communicate with any radio, as long as it respects the IDL constraints.
We tested platform independence by running the SDR’10 Demo using two machines with different operating systems – one was running a Mac OS X 10.6, and the other one, Ubuntu 10.04. Moreover, the radios were implemented in two different programming languages – Java and Python.

In summary, our conclusion was that CRF satisfies the requirement of platform independence as stated in Section 2.2.

5.2.4 Domain-adaptability

Domain-adaptability requires that the CR architecture can adapt to new SDR concepts and technology. These changes can be manifested in two different ways: (1) the CR ontology is redesigned or augmented, (2) SDRs offer new knobs and meters. A domain adaptable architecture can support these changes without requiring to recode the APIs in response to such changes in the domain.

The burden of adjusting to changes in the CR Ontology is passed to SDR vendors and rule writers. SDR vendors must update their IDL annotations and rule writers must adjust their rules. The architecture itself remains intact, because the reasoner – the component responsible for processing rules – and LiveKB – the component responsible for accessing radio’s parameters – are generic. This feature can be utilized to use CRF in domains entirely unrelated to CR.

5.2.5 Applying LiveKB in other domains

In this work we focused only on the CR architecture which makes use of an inference engine to provide the cognitive capability. The domain knowledge is provided in the form of ontologies and rules. As explained in Section 1.4, CR can be viewed as a subclass of a larger class of self-controlling software. Since the CRF architecture is domain-adaptable, it should be
possible to utilize it in other domains, where the self-controlling software paradigm applies. In particular, the LiveKB component, which translates the abstract ontological terms into software-specific invocations, could be configured for use in an entirely different domain than CR.

Figure 5.2 shows the architecture of a generic self-controlling software system which uses the LiveKB component. Analogously to CRF, LiveKB and the reasoner need to be provided with the plant-specific information. The reasoner needs to be configured with the domain ontology and rules, and LiveKB requires the plant’s annotated IDL model and a domain ontology. Moreover, as in the case of SDRs, the plant needs to make itself available through CORBA. Note that this design was created by simply removing SDR-related components from CRF and leaving only those that are truly generic. When applying this design to other domains, new components may be needed.

Figure 5.2: Application of LiveKB in self-controlling software
Using LiveKB in multisensor radar tracking

Multisensor data fusion techniques combine data from multiple sensors and related information, to achieve improved, more accurate and specific inferences than could be achieved by the use of a single, independent sensor. Data fusion is naturally performed by humans and animals, which combine information coming from their senses in order to improve their ability to survive, e.g. identify threats in the surrounding environment [115]. Multisensor radar tracking is an extensively used military application of the multisensor data fusion. Sensor suites may include radars, sonars, infrared, identification of friend or foe systems, or electronic support measures data. Using observations from multiple sensors, the tracker can perform the following tasks [116]:

a) estimate the state of a moving object — its position, heading and velocity
b) estimate the track’s identify
c) analyze the target’s intent

The first task is typically carried out by sequential estimation techniques such as Kalman Filter or its variants [116]. The second includes pattern recognition techniques, neural networks, or decision-based methods. The knowledge-based systems, suitable for exploiting both explicit and implicit information, are employed in implementing the third task, the threat analysis.

Similarly to the SDR domain, there are no standards for implementing the trackers. Research in this field has been mostly focused on improving existing algorithms, e.g. [117]. However, similarly to SDR, there is a range of contextual information (e.g. weather conditions) that could be taken advantage of to improve the quality of the tracking performance. Likewise, the problem arises when trying to implement a cognitive tracker and interface it with existing software.
In order to show that the LiveKB component can be adapted to the tracking domain we developed a simple application that made use of LiveKB and followed the design from Figure 5.2. First, we adapted the Kalman Filter Matlab\textsuperscript{6} code to Java and built an interface for it (see Figure 5.3). We specified some knobs and meters and made them available via an annotated IDL.

![Kalman Filter Demo](image)

**Figure 5.3:** Screenshot of a Kalman Filter demo

The LiveKB component was configured with a very basic Kalman Filter ontology and an IDL model. To demonstrate the use of the knobs and meters we wrote a rule which arbitrarily changed a value of one knob whenever a value of some meter was at a particular level. We also provided a graphical user interface which allowed for direct modification of the

\textsuperscript{6}Matlab source code is available at: http://www.innovatia.com/software/papers/kalman.m
meter values in order to simulate changes in the environment. The LiveKB implementation was successfully adapted — it produced proper ABox and translated the set requests into specific method invocations.

5.2.6 Summary

Table 5.1 summarizes how CRF supports the requirements for CR architecture, or how it meets the goals set forth in this thesis. Most of them are fulfilled, although SDRs do have to meet the requirements of LiveKB, which in some cases may be too constraining. As long as they can be realized, CRF offers better flexibility than any other radio interface used in the reviewed architectures. Moreover, it does not require introducing a radio-specific standard.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reusable knowledge</td>
<td>Full</td>
<td>With the use of CRF, the same rules and ontologies can be reused across numerous SDR implementations. Additionally, radios can exchange rules, as long as they are written in reference to a common ontology.</td>
</tr>
<tr>
<td>SDR-Independent</td>
<td>Partial</td>
<td>CRF supports a wide range of SDRs with different APIs, as long as they meet the architectural constraints specified by LiveKB.</td>
</tr>
<tr>
<td>Platform-Independent</td>
<td>Full</td>
<td>CRF’s support for platform independence relies on CORBA as the underlying middleware.</td>
</tr>
<tr>
<td>Domain-adaptable</td>
<td>Full</td>
<td>Changes in the domain (ontology or API) must be reflected in the artifacts provided by SDR vendors (annotations) and rule writers (rule updates), but they do not require recoding of any part of CRF.</td>
</tr>
</tbody>
</table>
5.3 Comparison of domain-API based and LiveKB based interfaces

In order to evaluate the costs and benefits of using the CRF approach to the integration of a reasoner with domain software, we give a more in-depth analysis of the comparison between the LiveKB approach and other approaches reviewed in Section 2.3. Since all of the other approaches rely on non-standard domain-specific interfaces we limit our analysis to one architecture which is an abstraction of all of the other approaches. We refer to this architecture as the domain-API based architecture. We implemented this architecture for the sake of this comparison.

Figure 5.4 shows the domain-API based architecture and CRF side by side. In both cases, “Domain Software” is used rather than SDR to stress the fact that the architectures are domain independent. In both cases, the reasoner monitors and controls the Domain Software. The reasoner’s KB is populated by some controller with ABox facts representing the current state of the domain software. When the reasoner is started, it may fire some rules, which in turn may make use of procedural attachments, which modify software parameters.

5.3.1 Domain adaptability and developer’s effort

The following scenarios describe changes that might occur in a particular domain. These changes may or may not require developer’s action to maintain an interface between a reasoner and the domain software.

**Scenario 1**: Ontology has been redesigned – the class hierarchy has changed, some entities changed their names, available knobs and meters remained the same.

**Scenario 2**: Ontology has been augmented to include new parameters (knobs and meters) available in software.
**Scenario 3**: Switch to a new domain – ontology, rules and domain software have all been replaced.

**Scenario 4**: Domain software API has changed to a new version, not backwards compatible with its predecessor.

Table 5.2 shows the types of developer’s effort required to integrate a reasoner using both a domain-API and LiveKB-based design. It is assumed that domain software is available via CORBA, and its IDL model follows the constraints required by LiveKB. We can see that in the approach based on LiveKB the programmer’s effort is limited to developing appropriate annotations. This requires the programmer gain an understanding of both the ontology used and of the domain software. However, the modification (annotations) are limited to inserting comment lines in the domain software. For the domain-API approach, on the other hand, significant code development is involved to achieve the same task. Consequently, it is safe to
Table 5.2 Comparison of developer’s effort required to adjust to changes in different scenarios

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Domain-API</th>
<th>LiveKB interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario 1</td>
<td>Rewrite code responsible for creating those ABox individuals that are instances of the new classes</td>
<td>Adjust annotations in IDL to provide proper property chains. <em>No recoding is necessary.</em></td>
</tr>
<tr>
<td>Scenario 2</td>
<td>(1) Develop new procedural attachments to make use of the new knobs &amp; meters, (2) Add code responsible for generating facts corresponding to new parameters</td>
<td>Add IDL annotations to the new methods. <em>No recoding is necessary.</em></td>
</tr>
<tr>
<td>Scenario 3</td>
<td>(1) Implement new domain API, (2) develop new procedural attachments, (3) write the code responsible for creating ABox</td>
<td>Annotate IDL for the domain ontology. <em>No recoding is necessary.</em></td>
</tr>
<tr>
<td>Scenario 4</td>
<td>(1) Either implement an adapter (if it is possible), or implement domain API from scratch, (2) Update procedural attachments, (3) Update code responsible for producing ABox</td>
<td>Move annotations to the new IDL. <em>No recoding is necessary.</em></td>
</tr>
</tbody>
</table>

state that the LiveKB based approach is requires less effort and thus can be claimed more (or easier) adaptable to changes in the domain.

5.3.2 Run-time complexity

One of the design goals of LiveKB was to ensure that the run-time complexity of this approach is not prohibitively high. We analyze this issue in two steps. First we list the main operations that need to be performed by each approach in the process of interacting with a reasoner. These operations are shown in Table 5.3. The Bootstrap operation is executed only once, when the interface is established. getAll is executed each time the reasoner is restarted to get the current values of knobs and meters. set is executed from within a procedural attachment when a particular rule is triggered to change a parameter’s value. get can be used to retrieve a value of a single parameter on demand, however, as was stated earlier, it will not be used from within the rules in the current implementation.

Because LiveKB uses reflection to access software’s parameters, it cannot simply invoke a
method, it must first get hold of the reference to the object and the method that give access
to the parameters. The biggest difference between the two approaches occurs at bootstrap.
Since LiveKB is generic, it must be configured for a particular software and this requires
several steps. Although this plays to LiveKB’s disadvantage, this operation is performed
only once, when the application is started.

Table 5.4 shows the run-time complexity of running the major interface operations de-
scribed above. Except for the bootstrap operation, it is roughly the same for both domain-
API and LiveKB approaches. When using LiveKB, the bootstrap operation has complexity
\( O(i \cdot m \cdot c) \) since it comprises several steps, proportional to the number of IDL interfaces, the
number of methods and attributes per interface and the length of the annotations, respect-
ively. The domain-API complexity is \( O(1) \) (the interface is simply instantiated). Therefore
the LiveKB complexity is higher. However, the bootstrap operation is performed only once,
when the interface is created, and thus it does not have a significant impact on the overall complexity of this approach.

Table 5.4 Comparison of run-time complexity

<table>
<thead>
<tr>
<th>Method</th>
<th>Domain interface</th>
<th>LiveKB interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bootstrap</td>
<td>$O(1)$</td>
<td>$O(i \cdot m \cdot c)$, $i$—number of IDL interfaces, $m$—number of methods and attributes in one interface, $c$—length of the domain property chain related to a method/attribute</td>
</tr>
<tr>
<td>getAll</td>
<td>$\Theta(n)$, $n$—number of getter methods</td>
<td>$\Theta(n)$, $n$—number of getter methods</td>
</tr>
<tr>
<td>get</td>
<td>$O(1)$</td>
<td>$O(1)^a$</td>
</tr>
<tr>
<td>set</td>
<td>$O(1)$</td>
<td>$O(1)^a$</td>
</tr>
</tbody>
</table>

$^a$Before a specific getter method is invoked, the right object and method are first found in a hash table. Searching for an element in a hash table may theoretically take as long as $O(n)$. In practice, the average time is $O(1)$.

More details of these operations for LiveKB is included in Appendix B, where each operation is represented in pseudocode.

5.3.3 Constraints and requirements

Table 5.5 shows the comparison of constraints and requirements posed on the domain software for each of the two approaches. When the domain-API approach is used, software is required to implement a concrete getter and setter API compatible with the reasoner and a domain controller. If the API expected by the reasoner is not provided, adapters must be implemented and maintained.

In the LiveKB approach, software developers have more freedom in how they design their API, provided the API satisfies the LiveKB constraints, as specified in Section 5.2.2.

Table 5.5 Comparison of requirements

<table>
<thead>
<tr>
<th>Feature</th>
<th>Domain-API</th>
<th>LiveKB interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>API/IDLModel</td>
<td>Strictly defined, may require adapters</td>
<td>Arbitrary within LiveKB constraints</td>
</tr>
<tr>
<td>Middleware</td>
<td>CORBA (optional)</td>
<td>CORBA and LiveKB</td>
</tr>
</tbody>
</table>
5.4 Increase in space complexity due to IDL ontology and bridge axioms

The LiveKB approach requires a representation of a model of domain software in IDL. Moreover, this model must be expressed in an ontology. This leads to an increase in the number of logical clauses (RDF triples) in the reasoner’s knowledge base. In this section we assess the level of this increase.

Additional triples produced by using LiveKB are related to the length of an IDL document and to the number and length of annotations of the IDL constructs. LiveKB produces an IDL ontology that reflects part of the IDL model. Table 5.6 shows the number of triples for each IDL construct represented in OWL.

<table>
<thead>
<tr>
<th>IDL construct</th>
<th>Produced triples</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface I</td>
<td>(I rdf:type owl:class)</td>
<td>1</td>
</tr>
<tr>
<td>interface I : S</td>
<td>(I rdf:type owl:class) (I rdfs:subClassOf S)</td>
<td>2</td>
</tr>
<tr>
<td>attribute type name</td>
<td>(name rdf:type owl:DatatypeProperty) (name rdf:type owl:FunctionalProperty) (name rdfs:domain I) (name rdfs:range type) (name rdfs:subPropertyOf idl:getterMethod) (name rdfs:subPropertyOf idl:setterMethod)</td>
<td>6</td>
</tr>
<tr>
<td>context: interface I, type is primitive</td>
<td>(name rdf:type owl:DatatypeProperty) (name rdf:type owl:FunctionalProperty) (name rdfs:domain I) (name rdfs:range type) (name rdfs:subPropertyOf idl:getterMethod) (name rdfs:subPropertyOf idl:setterMethod)</td>
<td>5</td>
</tr>
<tr>
<td>context: interface I, type is primitive</td>
<td>(name rdf:type owl:DatatypeProperty) (name rdf:type owl:FunctionalProperty) (name rdfs:domain I) (name rdfs:range type) (name rdfs:subPropertyOf idl:getterMethod)</td>
<td>5</td>
</tr>
<tr>
<td>void name(out type value)</td>
<td>(name rdf:type owl:DatatypeProperty) (name rdf:type owl:FunctionalProperty) (name rdfs:domain I) (name rdfs:range type) (name rdfs:subPropertyOf idl:getterMethod)</td>
<td>5</td>
</tr>
<tr>
<td>void name(in type value)</td>
<td>(name rdf:type owl:DatatypeProperty) (name rdf:type owl:FunctionalProperty) (name rdfs:domain I) (name rdfs:range type) (name rdfs:subPropertyOf idl:setterMethod)</td>
<td>5</td>
</tr>
</tbody>
</table>

In order to construct distinct datatype property chains, LiveKB must assert several new
axioms to the ontology. For each class that appears in the middle of the chain, it has to create a new functional object property and define a self-restriction on it. Thus, for each class that appears in the middle of the IDL annotation, the following 8 triples are asserted:

\begin{align*}
  \text{is}_\text{Class} & \quad \text{rdf:} \text{type} & \quad \text{owl:ObjectProperty} \\
  \text{is}_\text{Class} & \quad \text{rdf:} \text{type} & \quad \text{owl:FunctionalProperty} \\
  \text{is}_\text{Class} & \quad \text{rdfs:} \text{domain} & \quad \text{Class} \\
  \text{is}_\text{Class} & \quad \text{rdfs:} \text{range} & \quad \text{Class} \\
  :R & \quad \text{rdf:} \text{type} & \quad \text{owl:Restriction} \\
  :R & \quad \text{owl:onProperty} & \quad \text{is}_\text{Class} \\
  :R & \quad \text{owl:} \text{hasSelf} & \quad \text{true} \\
  \text{Class} & \quad \text{owl:} \text{equivalentClass} & \quad :R
\end{align*}

where \( :R \) indicates an anonymous restriction class. Moreover, for each property \( p \) in the chain (including the locally reflexive properties) the following three triples are asserted to form an ordered list:

\begin{align*}
  :s & \quad \text{rdf:} \text{type} & \quad \text{rdf:List} \\
  :s & \quad \text{rdf:} \text{first} & \quad p \\
  :s & \quad \text{rdf:} \text{rest} & \quad np
\end{align*}

where \( :s \) indicates a blank node. If \( p \) is the last property in the chain, \( np \) is \( \text{rdf:nil} \), otherwise it is the name of the next property in the chain. Finally, one more triple is asserted to
indicate that the chain is a subproperty of a property in the IDL ontology:

```
idl:property owl:propertyChainAxiom _:h
```

where _:h indicates the head of the list that represents the property chain.

**Example**

To better illustrate how many triples are asserted for property chains, we show an example. Consider the following chain:

```
Class1.objectProperty1.Class4.objectProperty2.Class5*.datatypeProperty
```

We need to create self-restrictions on new properties `is_CLASS4` and `is_CLASS5`, which takes 8 triples each. The property chain then looks like this:

```
objectProperty1 ◦ is_CLASS4 ◦ objectProperty2 ◦ is_CLASS5 ◦ dataProperty
```

For each of the 5 properties in the chain, 3 triples are created to form a list element. Finally, we need to assert that the chain is a subproperty of a particular IDL property. Thus, asserting information about this chain requires a total of 32 triples:

```
2 * 8 + 5 * 3 + 1 = 32
```

In summary, the number of additional triples is on the order of $O(i \cdot m \cdot c)$, since it is proportional to the number of IDL interfaces, the number of getters and setters per interface and the length of the annotations, respectively. However, the number of triples is constant during software’s operation, because these are the TBox triples only. These additional triples
are created only at bootstrap and their number does not change. Perhaps in some cases it may be more efficient to run a separate reasoner within LiveKB, thus allowing the rules to be more succinct and not produce additional triples in the domain reasoner’s KB.

5.4.1 Summary

The benefits of using LiveKB interface to access domain software’s parameters are: support for knowledge reusability and exchange, significantly smaller effort required to adapt to changes, inherent platform-independence. The drawbacks of using LiveKB include the requirement for using CORBA, increased number of triples in the reasoner’s KB, slightly longer rules and slower bootstrap.

We conclude that if the domain, in which the interface is applied, does not change often and is already well-standardized, using LiveKB might be an unnecessary burden. However, if the domain lacks standards and changes are likely to happen in the future (like in the wireless domain), LiveKB offers a very flexible mechanism for accessing software in such environment.
Chapter 6

Conclusion

6.1 Contributions

The following are the major contributions of this research:

- Identified the need for a universal, domain and platform-independent interface between SDRs and reasoners as an architectural requirement that is necessary for a successful implementation of Cognitive Radio.

- Developed the concept of an API that allows for access to radio’s knobs and meters using abstract ontological terms.

- Reviewed and analyzed various existing architectures with the focus on the interface with SDRs.

- Designed and implemented the LiveKB component that facilitates the concept of a universal interface with self-controlling software.

- Designed and implemented a Cognitive Radio Framework that supports knowledge reusability and exchange, SDR and platform-independence, and domain-adaptability.
An assessment of the implementation effort in terms of lines of code is shown in Table 6.1.

- Analyzed and compared the LiveKB approach against the use of domain API to access radio’s parameters.
- Augmented the implementation of the Alignment API to support generation of OWL property chains from ontology alignments expressed in EDOAL.
- Designed and implemented a finite state machine that incorporates three of the standard FIPA interaction protocols.

<table>
<thead>
<tr>
<th>Components</th>
<th>Approximate number of lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRF core</td>
<td>3,700</td>
</tr>
<tr>
<td>CRF GUI</td>
<td>1,200</td>
</tr>
<tr>
<td>Selective-Repeat protocol</td>
<td>1,000</td>
</tr>
<tr>
<td>LiveKB</td>
<td>2,200</td>
</tr>
<tr>
<td>Image Transceiver</td>
<td>900</td>
</tr>
<tr>
<td>Generic Controller</td>
<td>700</td>
</tr>
<tr>
<td>SDR platforms</td>
<td>1,300</td>
</tr>
<tr>
<td>Kalman Filter</td>
<td>900</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>11,900</strong></td>
</tr>
</tbody>
</table>

**6.2 Future Research**

The current implementation of CRF and LiveKB provides a starting point for further research in the knowledge-based approach to implementing CR. Future tasks may include:

- Improve the matching algorithm to support creating bridge axioms without requiring the IDL annotations.
• Increase the number of IDL models that are supported by LiveKB, specifically, include automatic support for operations with multiple parameters (using the Command design pattern) and for generating complex ABox graphs.

• Develop effective ways to coordinate FIPA interaction protocols with rules, including support for interactions with multiple radios.

• Investigate different methods to initiate knowledge exchange between heterogeneous radios, e.g., use pilot channels.

• Develop additional use cases involving knowledge-exchange that utilize CRF.

• Analyze existing reasoner APIs and select one that is most advantageous to the CR domain.

• Investigate use of Web Services as a standard SDR API.

• Develop effective design to interface CRF with SCA-based radios.
Bibliography


Appendices
Appendix A

FIPA FSM

In this appendix we present the state machine that was implemented within the CRF. Figure A.1 shows the top-level view of the FSM with all possible states that the radio can transition to when in the IDLE state. If the radio receives a message from another radio, it becomes a Participant of an interaction with another agent (radio). The type of the received message will determine which state the radio transitions to. If, however, the radio does not receive any message for some time (defined in configuration), an initEvent is automatically invoked and the reasoner is triggered. If the reasoner generates an ACL message that needs to be sent over to another radio, it becomes that Initiator of the interaction.

The FIPA standard includes the following interaction protocols: Request, Query, Request When, Contract Net, Iterated Contract, English Auction, Dutch Auction, Brokering, Recruiting, Subscribe, Propose. All of them are described using AUML[80] sequence diagrams, however, these diagrams are not sufficient to implement multiple protocols at once, or how to handle exceptions. For this reason, we designed a FSM that supports the following standard Interaction protocols:

- **Query** (Figure A.2) – allows radios to query each other’s KB

- **Request** (Figure A.3) – allows radios to send commands to each other, e.g. increase
transmission power.

- **Contract Net** (Figure A.4) – this rather complex protocol can be used to allow radios to query each other for proposals. For instance, a radio can call for proposal regarding new channel parameters.

Extending the FSM to support more interaction protocols would require extending the **IDLE** and **Running Rules** states, so that new types of messages are recognized.

![Diagram of the FIPA FSM](image)

**Figure A.1**: Top-level view of the FIPA FSM.
Figure A.2: FIPA Query protocol.
Figure A.3: FIPA Request protocol.
Figure A.4: FIPA Contract Net protocol.
Appendix B

Algorithms used in LiveKB

In this appendix we present pseudocode for the algorithms implemented within the LiveKB component. For each algorithm we also provide the asymptotic running time, using the notational conventions from [118]. The asymptotic analysis of algorithms is an important part of the evaluation of the CRF architecture.

The algorithms listed below correspond to the LiveKB API and comprise:

- **Bootstrap** (Algorithm B.1) – this corresponds to the `getInstance` method from the LiveKB API and is executed only once per operation, when the instance of LiveKB is created.

- **getAll** (Algorithm B.2) – this algorithm is used to create ABox facts and it makes use of artifacts created by Bootstrap

- **get** (Algorithm B.4) – corresponds to the method, which allows for retrieving value of a parameter by passing the name in the IDL ontology.

- **set** (Algorithm B.3) – corresponds to the method that is used when the reasoner requests LiveKB to change value of a parameter on the radio.
Algorithm B.1 LiveKB* bootstrap algorithm – this is executed only once, when LiveKB is instantiated via the LiveKBFactory

Complexity: $O(i mc)$, where

- $i$ – number of IDL interfaces
- $m$ – number of methods and attributes in one interface
- $c$ – length of the property chain above a method

1: procedure Bootstrap(IdlModel, DomainOntology, RootObjectID)
2:   rootClass ← class to which RootObjectID belongs to
3:   set rootClass as a root of the objectTree
4:   for each IDL Interface $I$ of IdlModel do
5:     ASSERT($I$, rdf:type, owl:Class)
6:     for each superclass $S$ of $I$ do
7:       ASSERT($I$, owl:subClassOf, $S$)
8:     end for
9:   if $I$ is not rootClass then
10:      objectTree.addNode($I$)
11:   end if
12:   for each IDL method or attribute $x$ of $I$ do
13:     $name$ ← $I$.name + @ + $x$.name
14:     (isGetter, isSetter, type) ← COLLECTTYPEINFORMATION($x$)
15:     if type is primitive then
16:       ASSERT($name$, rdf:type, owl:DatatypeProperty)
17:     else if type is an IDL Interface then
18:       ASSERT($name$, rdf:type, owl:ObjectProperty)
19:     end if
20:     ASSERT($name$, rdfs:domain, $I$)
21:     ASSERT($name$, rdfs:range, type)
22:     if isGetter = true then
23:       ASSERT($name$, rdfs:subPropertyOf, idl:GetterProperty)
24:       if type is an IDL Interface then
25:         objectTree.addEdge($I$, name, type)
26:       end if
27:     end if
28:     if isSetter = true then
29:       ASSERT($name$, rdfs:subPropertyOf, idl:SetterProperty)
30:     end if
if chain above x is proper in DomainOntology then
  if type is primitive then
    dataProperty ← remove last element from the chain
    ind ← ASSERTCHAIN(chain)
    L ← L + (name, ind, dataProperty, isGetter, isSetter)
  else if type is an IDL Interface then
    ASSERTCHAIN(chain)
  end if
  Edoal ← Edoal + (chain, subsumedBy, name)
end if
end for
(S,G,A) ← SAVESETTERSANDGETTERS(L, objectTree)
idlOntology ← idlOntology + EDOAL2OWL(Edoal)  \ Via AlignmentAPI
end for
return (idlOntology, S, G, A)
end procedure

procedure COLLECTTYPEINFORMATION(x)  \ Complexity O(1)
isGetter ← false
isSetter ← false
if x is an attribute then
  type ← type of x
  isGetter ← true
  if x is not readonly then
    isSetter ← true
  end if
end if
else
  if x has no parameters and return type of x is not void then
    type ← type of x
    isGetter ← true
  else if x has one parameter and return type of x is void then
    direction ← direction of parameter passing
    type ← type of the parameter of x
    if direction = IN then
      isSetter ← true
    else if direction = OUT then
      isGetter ← true
    end if
  end if
end if
return (isGetter, isSetter, type)
end procedure
procedure Assert(s, p, o)  ▷ Complexity O(1)
idlOntology ← idlOntology + (s, p, o)
end procedure

procedure AssertIndividual(ClassName)  ▷ Complexity O(1)
ind ← aClassName
if individual ind has not been created or is marked as new then
  Assert(ind, owl:type, ClassName)
  Assert(ind, is_ClassName, ind)
end if
return ind
end procedure

procedure AssertChain/owlChain)  ▷ Complexity O(n), n - length of chain
for each (s, p, o) triple of the owlChain do
  Assert(s, rdf:type, owl:ObjectProperty)
  Assert(s, rdf:domain, o)
  Assert(s, rdf:range, o)
  Assert(o, rdfs:subClassOf, ObjectHasSelf(is_o))  ▷ Reflective identity property
dom ← AssertIndividual(s)
range ← AssertIndividual(o)
Assert(dom, p, range)
end for
return range
end procedure

procedure SaveSettersAndGetters(L, objectTree)
Complexity O(md), where:
  • m – number of getters and setters in the IDL
  • d – object tree depth
  
for i = 0 to L.length do
  (idlMethodName, individual, property, isGetter, isSetter) ← L[i]
  (object, method) ← FINDMETHOD(objectTree, idlMethodName)
  if isSetter then
    setters.put(idlMethodName, (object, method))
  end if
  if isGetter then
    getters.put(idlMethodName, (object, method))
    A ← A + (individual, property, object, method)
  end if
end for

return (setters, getters, A)
end procedure
procedure FindMethod(objectTree, idlMethodName)

Complexity $O(d)$, $d$ - tree depth

idlClassName ← idlMethodName.substringBefore(@)

path ← objectTree.getPathTo(idlClassName)

method ← remove last element from the path

object ← objectTree.getRootObject

for each m in the path do

object ← object.m

end for

return (object, method)

end procedure

Algorithm B.2 LiveKB getAll algorithm – this is executed each time a reasoner is restarted to get the current values of the ABox

1: procedure GetAll(A, Ontology) ⊲ Complexity $O(g)$, $g$ - number of getters
2: for i = 0 to A.length do
3:   (individual, property, object, method) ← A[i]
4:   value ← object.method()
5:   Ontology ← Ontology + (individual, property, value)
6: end for
7: return Ontology
8: end procedure

Algorithm B.3 LiveKB set algorithm – this is executed from within reasoner’s procedural attachments to change a value of a knob

1: procedure Set(S, idlMethodName, newValue) ⊲ Complexity $O(1)$
2: (object, method) ← S.get(idlMethodName)
3: object.method(newValue)
4: end procedure

Algorithm B.4 LiveKB get algorithm – this could potentially be executed from the reasoner’s procedural attachments to get a value of a knob

1: procedure Get(G, idlMethodName) ⊲ Complexity $O(1)$
2: (object, method) ← G.get(idlMethodName)
3: return object.method()
4: end procedure
Appendix C

Example code generated by LiveKB

When LiveKB is bootstrapped, it parses the annotated IDL to generate (1) **IDL ontology** and (2) an **EDOAL alignment document**. Using modified version of the Alignment API, Assisted Matcher translates EDOAL correspondences to (3) **OWL bridge axioms**. Each time the reasoner is restarted, it requests LiveKB to fetch the values of all radio’s parameters. LiveKB uses artifacts produced at bootstrap and generates (4) **OWL ABox axioms**, which represent the current state of the radio’s operational parameters. Once the reasoner’s KB is populated with the bridge axioms, IDL ontology, CR ontology’s TBox and ABox axioms, it can process the CR rules, which make use of the correspondences extracted by LiveKB from the IDL annotations.

In this appendix we list all of the four documents that are produced by LiveKB during its interactions with one SDR. The first three are produced only once, at bootstrap, and the last one is produced each time the reasoner requests values of all parameters.

The code listed below was generated after interactions between LiveKB and a mockup of an SDR that implemented the IDL interface presented in Listing C.1. The reference to the implementation of the root interface, TestRadio, was available under the name radioA in the CORBA Naming Service.
Listing C.1  Annotated IDL implemented by a mockup SDR used as an example

```idl
module api {
  interface SignalDetector {
    // Radio.hasSubComponent.SignalDetector.componentName
    readonly attribute string id;
    // Radio.hasSubComponent.SignalDetector.signalToNoiseRatio.
    // Decibel*.hasValue.FloatValue*.hasFloat
    readonly attribute float signalToNoiseRatio;
    // Radio.hasSubComponent.SignalDetector.sampleRate.SampleRate*.
    // hasValue.FloatValue*.hasFloat
    attribute float sampleRate;
  };

  interface Power {
    // hasValue.FloatValue*.hasFloat
    float getPower();
  };

  interface Transmitter {
    readonly attribute Power power;
    long getTransmitCycle();
    void setTansmitCycle(in long newTransmitCycle);
  };

  interface TestRadio {
    // Radio.componentName
    readonly attribute string radioId;
    readonly attribute Transmitter transmitter;
    readonly attribute SignalDetector signalDetector;
    float getTxAmplitude();
    void setTxAmplitude(in float newTxAmp);
  }
```

During bootstrapping, LiveKB produced the following:

1. IDL Ontology representing the IDL model (Listing C.2)
2. EDOAL alignment document created based on the annotations in the IDL (Listing C.3)
3. OWL bridge axioms generated from EDOAL using a modified version of the Alignment API (Listing C.4)

Once bootstrapped, LiveKB was used to generate ABox. Listing C.5 shows ABox with values of parameters that came from the mockup implementation of TestRadio.

**Listing C.2** Ontology generated from the TestRadio.idl

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns="http://ece.neu.edu.crf/SDROntology.owl#"
xml:base="http://ece.neu.edu.crf/SDROntology.owl"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:owl="http://www.w3.org/2002/07/owlX#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:idl="http://ece.neu.edu/livekb/IDLGenOnt.owl#"
xmlns:sdr="http://ece.neu.edu.crf/SDROntology.owl#">

<owl:Ontology
    rdf:about="http://ece.neu.edu.crf/SDROntology.owl"/>

<!-- OBJECT PROPERTIES -->
<owl:ObjectProperty rdf:about="sdr:is_Decibel">
    <rdf:type rdf:resource="owl:FunctionalProperty"/>
    <rdfs:range rdf:resource="sdr:Decibel"/>
    <rdfs:domain rdf:resource="sdr:Decibel"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="sdr:is_FloatValue">
    <rdf:type rdf:resource="owl:FunctionalProperty"/>
    <rdfs:range rdf:resource="sdr:FloatValue"/>
    <rdfs:domain rdf:resource="sdr:FloatValue"/>
</owl:ObjectProperty>
```
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="sdr:is_Power">
  <rdf:type rdf:resource="owl:FunctionalProperty"/>
  <rdfs:range rdf:resource="sdr:Power"/>
  <rdfs:domain rdf:resource="sdr:Power"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="sdr:is_PowerAmplifier">
  <rdf:type rdf:resource="owl:FunctionalProperty"/>
  <rdfs:domain rdf:resource="sdr:PowerAmplifier"/>
  <rdfs:range rdf:resource="sdr:PowerAmplifier"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="sdr:is_SampleRate">
  <rdf:type rdf:resource="owl:FunctionalProperty"/>
  <rdfs:range rdf:resource="sdr:SampleRate"/>
  <rdfs:domain rdf:resource="sdr:SampleRate"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="sdr:is_SignalDetector">
  <rdf:type rdf:resource="owl:FunctionalProperty"/>
  <rdfs:domain rdf:resource="sdr:SignalDetector"/>
  <rdfs:range rdf:resource="sdr:SignalDetector"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="sdr:is_Transmitting">
  <rdf:type rdf:resource="owl:FunctionalProperty"/>
  <rdfs:domain rdf:resource="sdr:Transmitting"/>
  <rdfs:range rdf:resource="sdr:Transmitting"/>
</owl:ObjectProperty>

<!-- DATA PROPERTIES -->
<owl:DatatypeProperty rdf:about="idl:getterMethod"/>

<owl:DatatypeProperty rdf:about="idl:setterMethod"/>

<owl:DatatypeProperty rdf:about="idl:Power@getPower">
  <rdf:type rdf:resource="owl:FunctionalProperty"/>
  <rdfs:subPropertyOf rdf:resource="idl:getterMethod"/>
  <rdfs:domain rdf:resource="idl:Power"/>
  <rdfs:range rdf:resource="xsd:float"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:about="idl:SignalDetector@id">
  <rdf:type rdf:resource="owl:FunctionalProperty"/>
  <rdfs:subPropertyOf rdf:resource="idl:getterMethod"/>
  <rdfs:domain rdf:resource="idl:SignalDetector"/>
  <rdfs:range rdf:resource="xsd:string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="idl:SignalDetector@sampleRate">
  <rdf:type rdf:resource="owl:FunctionalProperty"/>
  <rdfs:subPropertyOf rdf:resource="idl:getterMethod"/>
  <rdfs:subPropertyOf rdf:resource="idl:setterMethod"/>
  <rdfs:domain rdf:resource="idl:SignalDetector"/>
  <rdfs:range rdf:resource="xsd:float"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="idl:SignalDetector@signalToNoiseRatio">
  <rdf:type rdf:resource="owl:FunctionalProperty"/>
  <rdfs:subPropertyOf rdf:resource="idl:getterMethod"/>
  <rdfs:domain rdf:resource="idl:SignalDetector"/>
  <rdfs:range rdf:resource="xsd:float"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="idl:TestRadio@amplifierId">
  <rdf:type rdf:resource="owl:FunctionalProperty"/>
  <rdfs:subPropertyOf rdf:resource="idl:getterMethod"/>
  <rdfs:domain rdf:resource="idl:TestRadio"/>
  <rdfs:range rdf:resource="xsd:string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="idl:TestRadio@getTxAmplitude">
  <rdf:type rdf:resource="owl:FunctionalProperty"/>
  <rdfs:subPropertyOf rdf:resource="idl:getterMethod"/>
  <rdfs:domain rdf:resource="idl:TestRadio"/>
  <rdfs:range rdf:resource="xsd:float"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="idl:TestRadio@radioId">
  <rdf:type rdf:resource="owl:FunctionalProperty"/>
  <rdfs:subPropertyOf rdf:resource="idl:getterMethod"/>
  <rdfs:domain rdf:resource="idl:TestRadio"/>
  <rdfs:range rdf:resource="xsd:string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="idl:TestRadio@setTxAmplitude">
  <rdf:type rdf:resource="owl:FunctionalProperty"/>
  <rdfs:subPropertyOf rdf:resource="idl:getterMethod"/>
  <rdfs:domain rdf:resource="idl:TestRadio"/>
  <rdfs:range rdf:resource="xsd:float"/>
</owl:DatatypeProperty>

179
<owl:DatatypeProperty rdf:about="idl:TestRadio@getTestValue">
  <rdf:type rdf:resource="owl:FunctionalProperty"/>
  <rdfs:subPropertyOf rdf:resource="idl:getterMethod"/>
  <rdfs:domain rdf:resource="idl:TestRadio"/>
  <rdfs:range rdf:resource="xsd:float"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="idl:Transmitter@getTransmitCycle">
  <rdf:type rdf:resource="owl:FunctionalProperty"/>
  <rdfs:subPropertyOf rdf:resource="idl:getterMethod"/>
  <rdfs:domain rdf:resource="idl:Transmitter"/>
  <rdfs:range rdf:resource="xsd:integer"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="idl:Transmitter@setTransmitCycle">
  <rdf:type rdf:resource="owl:FunctionalProperty"/>
  <rdfs:subPropertyOf rdf:resource="idl:setterMethod"/>
  <rdfs:domain rdf:resource="idl:Transmitter"/>
  <rdfs:range rdf:resource="xsd:integer"/>
</owl:DatatypeProperty>

<!-- CLASSES -->
<owl:Class rdf:about="sdr:Decibel">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="sdr:is_Decibel"/>
      <owl:hasSelf rdf:datatype="xsd:boolean">true</owl:hasSelf>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:about="sdr:FloatValue">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="sdr:is_FloatValue"/>
      <owl:hasSelf rdf:datatype="xsd:boolean">true</owl:hasSelf>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:about="sdr:Power">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="sdr:is_Power"/>
      <owl:hasSelf rdf:datatype="xsd:boolean">true</owl:hasSelf>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
Listing C.3  EDOAL alignment document generated from the annotated TestRadio.idl

```xml
<?xml version="1.0"?>
<owl:Class rdf:about="sdr:PowerAmplifier">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="sdr:is_PowerAmplifier"/>
      <owl:hasSelf rdf:datatype="xsd:boolean">true</owl:hasSelf>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:about="sdr:SampleRate">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="sdr:is_SampleRate"/>
      <owl:hasSelf rdf:datatype="xsd:boolean">true</owl:hasSelf>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:about="sdr:SignalDetector">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="sdr:is_SignalDetector"/>
      <owl:hasSelf rdf:datatype="xsd:boolean">true</owl:hasSelf>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:about="sdr:Transmitting">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="sdr:is_Transmitting"/>
      <owl:hasSelf rdf:datatype="xsd:boolean">true</owl:hasSelf>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

181
  <Alignment>
    <xml>yes</xml>
    <level>2EDOAL</level>
    <type>**</type>
    <onto1>
      <Ontology rdf:about="http://ece.neu.edu/livekb/IDLGenOnt.owl">
        <formalism>
          <Formalism align:name="OWL2.0"
            align:uri="http://www.w3.org/2002/07/owl#"/>
        </formalism>
      </Ontology>
    </onto1>
    <onto2>
      <Ontology rdf:about="http://ece.neu.edu.crf/SDROntology.owl">
        <formalism>
          <Formalism align:name="OWL2.0"
            align:uri="http://www.w3.org/2002/07/owl#"/>
        </formalism>
      </Ontology>
    </onto2>
    <map>
      <Cell>
        <entity1>
          <edoal:Property
            rdf:about="idl:Transmitter@getTransmitCycle"/>
        </entity1>
        <entity2>
          <edoal:Property
            rdf:parseType="Collection">
            <edoal:Relation
              rdf:about="sdr:participatesIn"/>
            <edoal:Relation
              rdf:about="sdr:is_Transmitting"/>
            <edoal:Property
              rdf:about="sdr:transmitCycle"/>
          </edoal:compose>
        </entity2>
      </Cell>
    </map>
  </Alignment>
</rdf:RDF>
<map>
  <Cell>
    <entity1>
      <edoal:Property
        rdf:about="idl:Transmitter@setTansmitCycle"/>
    </entity1>
    <entity2>
      <edoal:Property
        rdf:about="idl:Transmitter@setTansmitCycle"/>
    </entity2>
    <relation>Subsumes</relation>
    <measure rdf:datatype="xsd:float">1</measure>
  </Cell>
</map>

<map>
  <Cell>
    <entity1>
      <edoal:Property
        rdf:about="idl:Transmitter@setTansmitCycle"/>
    </entity1>
    <entity2>
      <edoal:Property
        rdf:about="idl:Transmitter@setTansmitCycle"/>
    </entity2>
    <relation>Subsumes</relation>
    <measure rdf:datatype="xsd:float">1</measure>
  </Cell>
</map>

<map>
  <Cell>
    <entity1>
      <edoal:Property
        rdf:about="idl:SignalDetector@id"/>
    </entity1>
    <entity2>
      <edoal:Property
        rdf:about="idl:SignalDetector@id"/>
    </entity2>
    <relation>Subsumes</relation>
    <measure rdf:datatype="xsd:float">1</measure>
  </Cell>
</map>
<map>
  <Cell>
    <entity1>
      <edoal:Property
        rdf:about="idl:SignalDetector@signalToNoiseRatio"/>
    </entity1>
    <entity2>
      <edoal:Property>
        <edoal:compose rdf:parseType="Collection">
          <edoal:Relation rdf:about="sdr:hasSubComponent"/>
          <edoal:Relation rdf:about="sdr:is_SignalDetector"/>
          <edoal:Relation rdf:about="sdr:signalToNoiseRatio"/>
          <edoal:Relation rdf:about="sdr:is_Decibel"/>
          <edoal:Relation rdf:about="sdr:hasValue"/>
          <edoal:Relation rdf:about="sdr:is_FloatValue"/>
          <edoal:Property rdf:about="sdr:hasFloat"/>
        </edoal:compose>
      </edoal:Property>
    </entity2>
    <relation>Subsumes</relation>
    <measure rdf:datatype="xsd:float">1</measure>
  </Cell>
</map>

<map>
  <Cell>
    <entity1>
      <edoal:Property
        rdf:about="idl:SignalDetector@sampleRate"/>
    </entity1>
    <entity2>
      <edoal:Property>
        <edoal:compose rdf:parseType="Collection">
          <edoal:Relation rdf:about="sdr:hasSubComponent"/>
          <edoal:Relation rdf:about="sdr:is_SignalDetector"/>
          <edoal:Relation rdf:about="sdr:sampleRate"/>
          <edoal:Relation rdf:about="sdr:is_SampleRate"/>
          <edoal:Relation rdf:about="sdr:hasValue"/>
          <edoal:Relation rdf:about="sdr:is_FloatValue"/>
          <edoal:Property rdf:about="sdr:hasFloat"/>
        </edoal:compose>
      </edoal:Property>
    </entity2>
    <relation>Subsumes</relation>
  </Cell>
</map>
<measure rdf:datatype="xsd:float">1</measure>
</Cell>
</map>

<map>
  <Cell>
    <entity1>
      <edoal:Property rdf:about="idl:Power@getPower"/>
    </entity1>
    <entity2>
      <edoal:Property>
        <edoal:compose rdf:parseType="Collection">
          <edoal:Relation rdf:about="sdr:participatesIn"/>
          <edoal:Relation rdf:about="sdr:is_Transmitting"/>
          <edoal:Relation rdf:about="sdr:nominalRFPower"/>
          <edoal:Relation rdf:about="sdr:is_Power"/>
          <edoal:Relation rdf:about="sdr:hasValue"/>
          <edoal:Relation rdf:about="sdr:is_FloatValue"/>
          <edoal:Property rdf:about="sdr:hasFloat"/>
        </edoal:compose>
      </edoal:Property>
    </entity2>
    <relation>Subsumes</relation>
    <measure rdf:datatype="xsd:float">1</measure>
  </Cell>
</map>

<map>
  <Cell>
    <entity1>
      <edoal:Property rdf:about="idl:TestRadio@radioId"/>
    </entity1>
    <entity2>
      <edoal:Property rdf:about="sdr:componentName"/>
    </entity2>
    <relation>SubsumedBy</relation>
    <measure rdf:datatype="xsd:float">1</measure>
  </Cell>
</map>

<map>
  <Cell>
    <entity1>
      <edoal:Property rdf:about="idl:TestRadio@getTxAmplitude"/>
    </entity1>
    <entity2>
      <edoal:Property rdf:about="idl:TestRadio@getTxAmplitude"/>
    </entity2>
    <relation>SubsumedBy</relation>
    <measure rdf:datatype="xsd:float">1</measure>
  </Cell>
</map>
<entity1>
<entity2>
<edoal:Property>
<edoal:compose rdf:parseType="Collection">
<edoal:Relation rdf:about="sdr:hasSubComponent"/>
<edoal:Relation rdf:about="sdr:is_PowerAmplifier"/>
<edoal:Property rdf:about="sdr:txAmplitude"/>
</edoal:compose>
</edoal:Property>
</entity2>
<relation>Subsumes</relation>
<measure rdf:datatype="xsd:float">1</measure>
</Cell>
</map>

<map>
<Cell>
<entity1>
<edoal:Property rdf:about="idl:TestRadio@setTxAmplitude"/>
</entity1>
<entity2>
<edoal:Property>
<edoal:compose rdf:parseType="Collection">
<edoal:Relation rdf:about="sdr:hasSubComponent"/>
<edoal:Relation rdf:about="sdr:is_PowerAmplifier"/>
<edoal:Property rdf:about="sdr:txAmplitude"/>
</edoal:compose>
</edoal:Property>
</entity2>
<relation>Subsumes</relation>
<measure rdf:datatype="xsd:float">1</measure>
</Cell>
</map>

<map>
<Cell>
<entity1>
<edoal:Property rdf:about="idl:TestRadio@amplifierId"/>
</entity1>
<entity2>
<edoal:Property>
<edoal:compose rdf:parseType="Collection">
<edoal:Relation rdf:about="sdr:hasSubComponent"/>
<edoal:Relation rdf:about="sdr:is_PowerAmplifier"/>
</edoal:compose>
</edoal:Property>
</entity2>
<relation>Subsumes</relation>
<measure rdf:datatype="xsd:float">1</measure>
</Cell>
</map>
Listing C.4 OWL bridge axioms generated from the EDOAL alignment using a modified version of Alignment API

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns="http://ece.neu.edu.crf/SDROntology.owl#"
    xml:base="http://ece.neu.edu.crf/SDROntology.owl"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owlX#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:idl="http://ece.neu.edu/livekb/IDLGenOnt.owl#"
    xmlns:sdr="http://ece.neu.edu.crf/SDROntology.owl#">
  <owl:Ontology rdf:about="http://ece.neu.edu.crf/SDROntology.owl"/>
  <owl:DatatypeProperty rdf:about="idl:SignalDetector@sampleRate">
    <owl:propertyChainAxiom rdf:parseType="Collection">
      <owl:ObjectProperty rdf:about="sdr:hasSubComponent"/>
      <owl:ObjectProperty rdf:about="sdr:is_SignalDetector"/>
      <owl:ObjectProperty rdf:about="sdr:sampleRate"/>
      <owl:ObjectProperty rdf:about="sdr:is_SampleRate"/>
      <owl:ObjectProperty rdf:about="sdr:hasValue"/>
      <owl:ObjectProperty rdf:about="sdr:is_FloatValue"/>
      <owl:DatatypeProperty rdf:about="sdr:hasFloat"/>
    </owl:propertyChainAxiom>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:about="idl:TestRadio@setTxAmplitude">
    <owl:propertyChainAxiom rdf:parseType="Collection">
      <owl:ObjectProperty rdf:about="sdr:hasSubComponent"/>
      <owl:ObjectProperty rdf:about="sdr:is_PowerAmplifier"/>
    </owl:propertyChainAxiom>
  </owl:DatatypeProperty>
</rdf:RDF>
```
<owl:DatatypeProperty rdf:about="sdr:txAmplitude"/>
</owl:propertyChainAxiom>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="idl:SignalDetector@id">
<owl:propertyChainAxiom rdf:parseType="Collection">
<owl:ObjectProperty rdf:about="sdr:hasSubComponent"/>
<owl:ObjectProperty rdf:about="sdr:is_SignalDetector"/>
<owl:DatatypeProperty rdf:about="sdr:componentName"/>
</owl:propertyChainAxiom>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="idl:TestRadio@radioId">
<rdfs:subPropertyOf>
<owl:DatatypeProperty rdf:about="sdr:componentName"/>
</rdfs:subPropertyOf>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="idl:TestRadio@getTxAmplitude">
<owl:propertyChainAxiom rdf:parseType="Collection">
<owl:ObjectProperty rdf:about="sdr:hasSubComponent"/>
<owl:ObjectProperty rdf:about="sdr:is_PowerAmplifier"/>
<owl:DatatypeProperty rdf:about="sdr:txAmplitude"/>
</owl:propertyChainAxiom>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="idl:Power@getPower">
<owl:propertyChainAxiom rdf:parseType="Collection">
<owl:ObjectProperty rdf:about="sdr:participatesIn"/>
<owl:ObjectProperty rdf:about="sdr:is_Transmitting"/>
<owl:ObjectProperty rdf:about="sdr:nominalRFPower"/>
<owl:ObjectProperty rdf:about="sdr:is_Power"/>
<owl:ObjectProperty rdf:about="sdr:hasValue"/>
<owl:ObjectProperty rdf:about="sdr:is_FloatValue"/>
<owl:DatatypeProperty rdf:about="sdr:hasFloat"/>
</owl:propertyChainAxiom>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="idl:Transmitter@setTransmitCycle">
<owl:propertyChainAxiom rdf:parseType="Collection">
<owl:ObjectProperty rdf:about="sdr:participatesIn"/>
<owl:ObjectProperty rdf:about="sdr:is_Transmitting"/>
<owl:DatatypeProperty rdf:about="sdr:transmitCycle"/>
</owl:propertyChainAxiom>
</owl:DatatypeProperty>
Listing C.5  OWL ABox generated by LiveKB based on the annotations provided in the TestRadio.idl
<owl:Ontology rdf:about="http://ece.neu.edu.crf/SDROntology.owl"/>

<owl:NamedIndividual rdf:about="sdr:radioAPowerAmplifier">
  <rdf:type rdf:resource="sdr:PowerAmplifier"/>
  <txAmplitude rdf:datatype="xsd:float">16.821407</txAmplitude>
  <componentName rdf:datatype="xsd:string">radioA_amplifierID</componentName>
  <is_PowerAmplifier rdf:resource="sdr:radioAPowerAmplifier"/>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="sdr:radioARadio">
  <rdf:type rdf:resource="sdr:Radio"/>
  <componentName rdf:datatype="xsd:string">radioA</componentName>
  <hasSubComponent rdf:resource="sdr:radioAPowerAmplifier"/>
  <is_Radio rdf:resource="sdr:radioARadio"/>
  <hasSubComponent rdf:resource="sdr:radioASignalDetector"/>
  <participatesIn rdf:resource="sdr:radioATransmitting"/>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="sdr:radioASignalDetector">
  <rdf:type rdf:resource="sdr:SignalDetector"/>
  <componentName rdf:datatype="xsd:string">radioA_signalDetectorID</componentName>
  <is_SignalDetector rdf:resource="sdr:radioASignalDetector"/>
  <signalToNoiseRatio>
    <Decibel>
      <hasValue>
        <FloatValue>
          <hasFloat rdf:datatype="xsd:float">6.1687</hasFloat>
          <is_FloatValue/>
        </FloatValue>
      </hasValue>
      <is_Decibel/>
    </Decibel>
    <SampleRate>
      <is_SampleRate/>
      <hasValue>
        <FloatValue>
          <hasFloat rdf:datatype="xsd:float">11.634969</hasFloat>
          <is_FloatValue/>
        </FloatValue>
      </hasValue>
    </SampleRate>
  </signalToNoiseRatio>
</owl:NamedIndividual>
</hasValue>
	</SampleRate>
</sampleRate>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="sdr:radioATransmitting">
	<rdf:type rdf:resource="sdr:Transmitting"/>
	<transmitCycle rdf:datatype="xsd:integer">30</transmitCycle>
	<is_Transmitting rdf:resource="sdr:radioATransmitting"/>
	<nominalRFPower>
		<Power>
			<hasValue>
				<hasFloat rdf:datatype="xsd:float">57.627094</hasFloat>
				<is_FloatValue/>
			</FloatValue>
			<is_Power/>
		</hasValue>
	</Power>
</nominalRFPower>
</owl:NamedIndividual>
</rdf:RDF>